MASTER EN SCIENCES MATHÉMATIQUES

Manipulation dynamique de valeurs en Algol 68

Vansina, Ch.

Award date:
1976

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES
NOTRE DAME DE LA PAIX
Institut d'Informatique

MANIPULATION DYNAMIQUE DE VALEURS EN ALGOL 68

1976

...s la direction de
J.P. CARDINAEL

Mémoire présenté pour l'obtention
du titre Maître en Informatique

Ch. VANSINA

FELIX QUI POTUIT RERUM
COGNOSCERE CAUSAS
Virgilius

Je tiens à remercier tout spécialement le Professeur J.P. CARDINAEL pour l'aide qu'il m'a fournie et surtout pour la patience qu'il a eu envers moi pour mener à bon terme ce mémoire.

Je remercie non moins vivement le Professeur GENNART et son équipe de chercheurs à l'ERM pour toute la collaboration et tous les bons conseils qu'ils m' ont donnés.

TABLE DES MATIERES

INTRODUCTION

a. GENERALITES

L'étude qui sera faite à l'occasion de ce mémoire, portera sur la manipulation dynamique de valeurs en ALGOL68.

Chaque programme ALGOL68 est un texte qui définit une séquence d'actions (fonctions primitives), dont la sémantique définit la signification du programme, et qui sont exécutables par un ordinateur. Les actions sont accomplies sur des objets internes. Chaque objet interne possède principalement trois qualificatifs :

1. Il est d'un certain mode
2. Il est une occurence particulière d'une valeur de ce mode
3. Cette valeur a un emplacement.

Le mode spécifie de quel matériel de base l'objet est constitué (bits, entiers, reels, etc ...). Certains modes définissent peu de valeurs (ex : le mode bool en admet deux : true ou false), d'autres en admettent une infinité (ex : le mode int admet tous les nombres entiers).

Il existe des modes de base (int, real, proc, ref, etc ...). A partir de ces modes on peut en définir d'autres qui, à leur tour peuvent servir de base pour de nouveaux modes. On peut facilement s'imaginer que potentiellement un mode peut devenir d'une complexité infinie. Les valeurs qui sont les occurences de ces modes ont le même degré de complexité. Pendant que l'ordinateur exécute un programme, des valeurs sont physiquement présentes en mémoire centrale. Parfois, il est nécessaire de les manipuler pendant l'exécution. Une de ces manipulations est l'obtention d'une copie d'une valeur de mode connu, se trouvant à un emplacement donné, vers un autre emplacement.

L'étude d'algorithmes, qui permettent de faire cette copie, sera le but principal de ce mémoire.

b. DESCRIPTION GLOBALE D'UN COMPILATEUR

Un compilateur est un programme qui traduit un programme écrit dans

un langage source, en un programme écrit dans un langage objet, en respectant intégralement la sémantique. Le langage source est généralement de haut niveau (ex : ALGOL68) et le langage objet est généralement du code machine. Habituellement, la traduction se fait en deux étapes l'analyse syntaxique et la traduction proprement dite.

L'analyse syntaxique est une transformation du programme source, qui met en évidence les fonctions primitives et leurs lois de compositions. Un analyseur syntaxique reçoit donc un texte source en entrée et produit une séquence de fonctions primitives qui respectent la sémantique en sortie.

La traduction proprement dite est l'action de produire des instructions dans un code machine à partir des fonctions primitives. Cette traduction peut se faire en plusieurs étapes. Chaque étape aura en sortie une séquence d'instruction ou de fonctions dans un langage intermédiaire. En agissant ainsi on peut espérer rester aussi longtemps que possible indépendant de la machine.

Une des fonctions résultant de l'analyse syntaxique est celle d'une copie de valeur pendant l'exécution.

La syntaxe de cette fonction (procédure) en PL/I à la forme suivante : COPY (ADDRS, ADDRØ, MØDE) ou

ADDRS : adresse de la valeur à copier (valeur source)
ADDRØ : adresse de la valeur copiée (valeur objet)
MØDE : mode de la valeur à copier.

L'introduction de la procédure copy dans le traducteur (ce qui n'est qu'un appel de procédure) aura comme résultat final, une séquence d'instructions en assembleur IBM 360 symbolique. L'exécution de ces instructions est l'action de copier une valeur.

L'utilisation du PL/I a deux buts :

1) donner l'algorithme de la copie dans un langage de haut niveau, indépendant de la machine.

2) montrer l'aptitude éventuelle du PL/I à servir de langage d'écriture de compilateur ou au moins de documentation d'un compilateur.

Le mémoire débouche pratiquement sur une routine de copie écrite en PL/I qui produit du code symbolique assembleur pour un ordinateur IBM 360.

Pour en arriver à ce stade on a choisi des objectifs intermédiaires qui peuvent être classés dans l'ordre suivant :

1) Approche théorique du problème et mise au point d'un algorithme global de génération de code.

2) Adaptation de cet algorithme à une configuration choisie de la gestion de la mémoire, pendant le déroulement d'un programme ALGOL68.

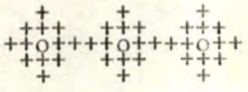
3) Description de l'algorithme en langage PL/I

REMARQUE

1) Dans ce qui suit, on trouve des programmes en PL/I et en ALGOL68. Le choix d'un des langages n'est pas aléatoire. Les programmes PL/I sont ceux qui décrivent l'algorithme de génération de code pour la copie. Les programmes ALGOL68 sont placés dans le texte comme exemples descriptifs du langage ALGOL68 lui-même.

2) Les programmes PL/I se trouvant dans le texte sont simplifiés en ce qui concerne la syntaxe. On y a enlevé les déclarations qui alourdissent inutillement la description de l'algorithme. On remplacera dans certains cas une séquence d'instructions par un texte français qui en décrit la sémantique.
Les programmes PL/I complets se trouvent en annexe.

3) L'implémentation choisie, ainsi que les tables utilisées pour tester l'algorithme ne sont pas purement sortis de l'imagination. Ils serviront d'éléments de base pour une implémentation future d'un compilateur ALGOL68 au centre informatique de l'ERM.
Le travail fait pour ce mémoire sera utilisé lors de cette implémentation.



CHAPITRE I : APPROCHE THEORIQUE DU PROBLEME

a. INTRODUCTION

Comme le problème soulevé correspond à une manipulation de valeurs de différents modes ALGOL68, on commencera par la représentation mémoire de ces valeurs. Une fois les représentations décrites, il est nécessaire de penser à une implémentation physique. C'est dans cette implémentation qu'on va décrire les détails de l'algorithme de copie.
Le problème de la copie d'une valeur mène à des procédés récursifs. Nous avons l'habitude de décrire des algorithmes sous forme d'ordinogramme. Un ordinogramme dans lequel on injecte des appels récursifs devient peu lisible. Pour cette raison, nous avons cru bon d'employer un moyen de description peu traditionnel, mais plus efficace. L'algorithme de génération de code pour copier une valeur est décrit par une grammaire de génération.

b. REPRESENTATION DES VALEURS ALGOL68

En ALGOL68 chaque valeur existante est d'un certain mode. On peut distinguer deux types de modes : les modes de base et les combinaisons obtenues à partir des modes de base.

Modes de base

Mode	*Valeur correspondante*
int (intiger)	valeur entière
real (real)	valeur réele
bool (boolean)	valeur booléene ou binaire
char (character)	caractère
bits	chaîne de valeurs binaires

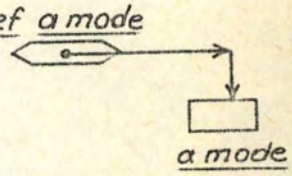
REMARQUE

On représente la valeur d'une mode de base par une case rectangulaire dans laquelle la valeur est inscrite.

Combinaison des modes de base

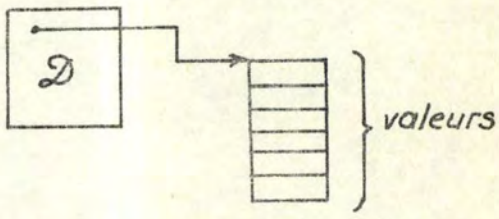
Il existe différents moyens de constituer de nouveaux modes à partir des modes de base.

ref amode : la valeur de ce mode est une référence (un nom) à une valeur de mode amode.
On représente un nom par une case comme suit:
La flèche représente le pointeur

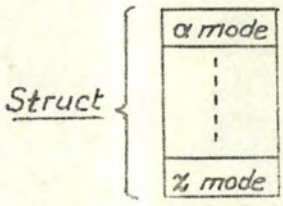


[] amode : Valeur multiple. La valeur se compose d'un nombre fini d'éléments de mode amode et d'un descripteur par lequel on accède à ces éléments.

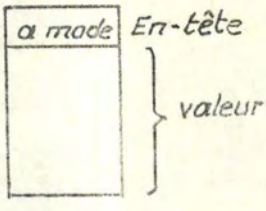
Représentation :



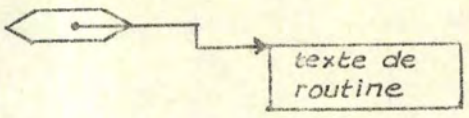
struct (amode,..., zmode) : Ensemble de valeurs qui sont respectivement de mode amode,..., zmode.



union (amode,..., zmode) : Valeur dont le mode peut se modifier pendant la durée de la vie. La valeur peut prendre les modes amode, zmode.
La valeur est constituée par un en-tête et la valeur même.
L'en-tête décrit la mode actuel de la valeur.



proc : La valeur de ce mode est une routine, c'est-à-dire un ensemble d'instructions ALGOL68 exécutables.



c. DEFINITIONS SUPPLEMENTAIRES

(1) *Portée*

La portée d'une valeur ALGOL68, est a morceau de programme dans lequel, pendant l'exécution, la valeur existe. Chaque valeur a une portée. Les valeurs simples (celles qui sont de mode int, real, bool, char, bits et bytes) ont comme portée tout le programme. D'autres valeurs qui sont, par leur mode, une référence à un autre mode, ont comme portée, le bloc dans lequel ils ont été déclarées. Ces valeurs sont celles de mode réf amode et toutes les valeurs dans lesquelles on peut retrouver une sous-valeur de mode ref amode.

Pendant l'exécution d'un programme, il est nécessaire que chaque valeur qui est de mode ref amode, ou qui en est composé, pointe vers une valeur qui a une portée égale ou plus large que sa propre portée. Illustrons cela part un exemple.

(a)
```
① begin int i ;
    ② begin int j=i ;
           i:= j ;
       end
end
```

(b)
```
① begin ref int xx ;
          int y ;
    ② begin int x ;
             x:=i;
             xx:=x;
       end
end
```

Dans le programme (a) il n'y a aucun problème, car la portée de la valeur que possède i, qui est de mode ref int est le bloc ① , la portée de la valeur que possède j est tout le programme (mode int). Après l'assignation i:=j ; la valeur que possède i pointe vers une valeur qui vit plus longtemps qu'elle-même.

Dans le programme (b) cependant, les valeurs ont les portées suivantes xx : bloc ① ; x = bloc ② . Après l'assignation xx : = x

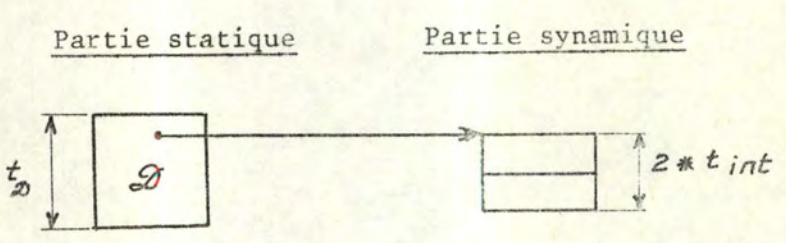
la valeur de xx pointe vers le nom possédé par x, qui a une portée plus petite que xx. En sortant du bloc (2) la valeur de x cesse d'exister, mais la valeur xx existe toujours et pointe vers une valeur non existante. Ceci n'est pas permis et doit être controlé (ou en reparle plus loin).

(2) *Partie dynamique d'une valeur*

Il existe des valeurs ALGOL68 qui se composent d'une partie statique et d'une partie dynamique. On appelle partie statique, la partie de la valeur dont la taille ne se modifie jamais. Donc une fois qu'on décide la création d'une valeur de ce mode on connaît l'encombrement de la partie statique (cet encombrement est fonction de l'implémentation). La partie dynamique est cette partie d'une valeur, qui peut avoir une taille variable d'une occurence à l'autre.

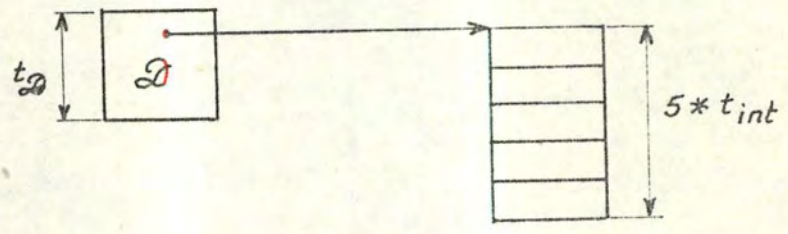
L'exemple typique d'une valeur qui admet une partie dynamique est la valeur de mode [] amode.

Exemple : [1 : 2] int = (1,2)



Descripteur

[1 : 5] int =(1,2,3,4,5)



Note : t_D = taille du descripteur, n'est que fonction de la dimension de la valeur multiple

t_{int} = taille d'une valeur de mode int

Il n'y a que trois modes qui peuvent avoir des valeurs possédant

une partie dynamique ; ce sont :[] amode ;

struct (amode a, ..., zmode z) ;

union (amode, ..., zmode) ;

Toutes les valeurs d'autres modes n'ont qu'une partie statique.

REMARQUE : Une valeur de mode struct ou union qui n'a pas de composantes de mode [] amode n'aura qu'une partie statique.

d. DESCRIPTION GENERALE DE L'ALGORITHME DE COPIE DE VALEURS

Le problème est de copier une valeur ALGOL68 d'un endroit connu de la mémoire, vers un autre endroit connu. Une copie de valeur est par exemple nécessaire dans les cas suivants ; assignations, passage de paramètres lors d'appels de procédure, passage des résultats à la sortie d'un bloc ou d'une routine, construction de "display" pour des valeurs multiples et pour des structures.

Le principe de base de la copie est fort simple : on copie la partie statique de la valeur en utilisant les adresses source et objet, après quoi, on copie la partie dynamique de la valeur.

La difficulté principale rencontrée en copiant, est le fait qu'une partie de la valeur objet peut surimprimer partiellement la valeur source. En implémentant les valeurs, selon des règles bien définies (1) on minimisera ces difficultés. L'implémentation sera étudiée en (2c (4) p 36)

Il y a des cas, où la surimpression ne peut être évitée parce que l'adresse source et l'adresse objet sont voisines. En tout cas, une partie statique objet ne peut jamais surimprimer une partie dynamique source. On décrit maintenant une stratégie qui résout ce problème de surimpression.

Procédé de la copie

On copie d'abord la partie statique de la valeur source, créant ainsi la partie statique de la valeur objet avec les anciens pointeurs. Si la valeur a une partie dynamique, on applique récursivement le procédé suivant : dans la partie statique objet on va rechercher tous les champs qui sont de mode [] amode. Pour chaque champ, on copie successivement les parties statiques des éléments de la valeur multiple et on met à jour le poin

---

(1) Ces règles sont étudiées en 2c (4) p 36.

teur dans le descripteur correspont. (partie statique de la partie dynamique)

Le fait de copier la partie dynamique d'une valeur à partir de sa partie statique objet, nous permet de surimprimer sans danger la valeur source correspondante.

Il faut toute fois faire attention de ne pas entamer la copie des parties statiques des éléments d'une valeur multiple avant qu'on soit certain qu'il y ait assez de place en mémoire pour le faire. De cette façon s'il manque de la place, les valeurs sources ne seront pas encore surimprimées au moment ou on se rend compte de ce manque de place et qu'on appele le "garbage collector". Quand on modifie le pointeur du descripteur de la valeur on sait qu'il pointera vers des valeurs existantes

Avant d'entamer la description formelle de l'algorithme de copie il faut encore décrire les actions faites en cas d'occurence de valeurs de modes ref amode. Dans la description ci-dessus on l'a volontairement omis pour ne pas compliquer l'algorithme. On dira qu'une valeur a une portée si elle est de mode ref amode ou si elle est de mode proc. Pour les valeurs d'autres modes ce ne sont pas les valeurs elle-mêmes qui ont une portée mais ce sont éventuellement leurs sous-valeurs qui en ont une. On justifiera l'idée de portée en décrivant l'implémentation des variables.

Pendant la copie d'une valeur, et après la copie de la partie statique on vérifie si la valeur à une portée. Si oui un test de portée s'effectue. Il faut que la portée de la valeur source soit, au moment de la copie, plus large ou égal à la portée de la valeur objet. Si ce n'est pas le cas, on arrête l'exécution du programme.

On peut décrire maintenant l'algorithme de la copie comme une séquence d'actions qui décrivent ce qui est fait pendant l'exécution du programme.

Action 1 : Copier la partie statique de la valeur source à l'adresse objet connue.

Action 2 : Si la valeur a une portée, effectuer le test de portée et ne plus faire d'actions pour cette valeur pendant l'exécution.
Si la valeur a une partie dynamique, ou qu'elle est de mode structurée, qui a au moins un champ ayant une portée passer à

1'action 3.
Sinon ne pas effectuer d'actions pendant 1'exécution.

Action 3 : Si la valeur est une structure, passer récursivement à 1'action 2 pour chacun de ses champs.
Si la valeur est une valeur multiple, passer à 1'action 4

Action 4 : 4a - Réserver de la place pour faire la copie de la partie statique des différents éléments de la valeur.

4b - S'il y a de la place, copier les parties statiques à la première adresse objet libre (la notion de première adresse libre sera expliquée lors de 1'implémentation).
S'il n'y a pas de place, appeler le "garbage collector" si on peut récupérer assez de place passer en 4b. Sinon, arrêter la copie et lancer un message d'erreur.

Note : Si les éléments d'une valeur multiple ne sont pas consécutifs en mémoire, ils sont compactés lors de la copie. Cela a deux avantages.

a) gain de place en mémoire
b) rendre les copies ultérieures plus efficaces

4c - Mettre à jour le pointeur du descripteur de la partie statique de la valeur traitée pendant 1'action 2 la plus récente.

4d - Passer récursivement à 1'action 2 pour chaque partie statique d'un élément de la valeur.

Pour rendre la compréhension de 1'algorithme plus aisée, on donne quelques exemples. Pour chaque exemple, on donne 1'état au départ et on donne la séquence de toutes les actions de 1'algorithme par lequel on passe. Chaque fois que quelque chose se modifie à 1'exécution de la copie, on donne le nouvel état de la valeur copiée.

Comme convention d'adressages ou prendra :

adresse source : adds
adresse objet : addo
le adresse libre : adde
adresse quelconque : addx

Exemple 1

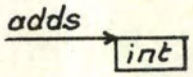
Mode de valeur : int

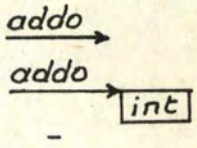
Actions de l'algorithme — *Valeur source* — *Valeur objet*

début

Action 1

Action 2





Exemple 2

Mode : struct (int i, ref int j) ;

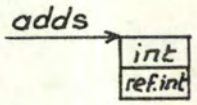
début

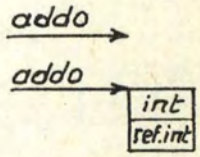
Action 1

Action 2

Action 3 champ int

champ ref int





test de portée

Exemple 3

Mode [1:5] int

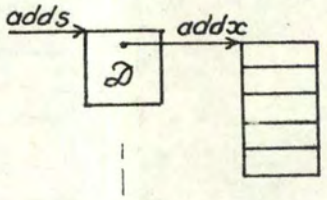
Action 1

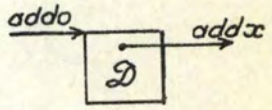
Action 2

Action 3

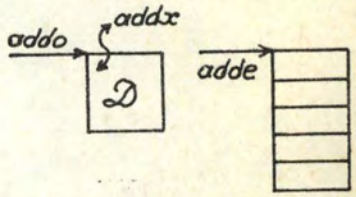
Action 4 : 4a

4b



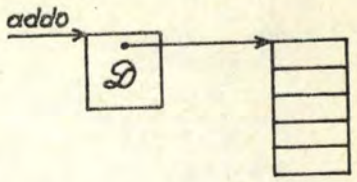


réserver de la place en addl



copie des parties statiques des éléments à la 1° adresse libre à partir de l'adresse addx.

4c



mise à jour du pointeur du descripteur

Pour chaque élément de la valeur on fait action 2 qui ne donne pas d'actions.

e. ALGORITHME GLOBAL DE GENERATION DE CODE

Jusqu'ici, on a étudié un algorithme qui permet de faire une copie d'une valeur pendant l'exécution du programme. Nous recherchons un algorithme qui, à partir d'une valeur nous génère un code symbolique dont l'exécution correspond parfaitement au déroulement de l'algorithme étudié ci-dessus.

Nous décrivons cet algorithme d'une façon originale. Habituellement, on décrit un procédé de travail à l'aide d'un organigramme. C'est un moyen valable pour autant qu'on ne veuille pas décrire un procédé récursif. A ce moment, un organigramme devient pratiquement illisible. Nous avons à traiter ici un procédé qui est complètement recursif. Pour en faire une description satisfaisante, nous avons cherché un autre moyen. Nous faisons la description de l'algorithme à l'aide d'une grammaire générative. On n'a pas l'habitude d'employer une grammaire pour décrire un procédé qui débouche sur un ensemble d'actions. Nous avons tenté notre chance, et nous sommes parvenu à décrire l'algorithme d'une façon claire et concise. Nous sommes même passé au-delà de notre but initial de décrire l'algorithme de la copie pendant l'exécution. Nous retrouvons en même temps les actions prises par le compilateur pour générer le code qui exécutera la copie.

En gros, cette grammaire se compose de terminaux et de non-terminaux. La chaîne terminale de sortie de la grammaire est l'image des actions prises pendant l'exécution. Les non-terminaux, par contre, correspondent à des actions prises par le compilateur pour pouvoir produire ces terminaux. A chaque terminal correspond un code généré bien défini.

On voit donc la puissance de ce moyen d'expression. On retrouve à

travers l'arbre grammatical les actions prises à deux niveaux différents : la compilation et l'exécution.

La grammaire que nous avons étudiée n'est pas ambigüe, il en résulte que la chaîne terminale peut être reconnue par la grammaire. On peut même, à partir de cette chaîne terminale reconstituer le mode de la valeur qu'on est en train de copier.

Une restriction est à faire : on ne fait aucune distinction dans la chaîne terminale entre les valeurs qui n'ont pas de partie dynamique. Pratiquement cela veut dire qu'il n'y a pas de différence entre le code généré pour faire la copie d'une valeur de mode int, real, bool, char, bits, bytes, struct (real, int, ... int), union (real, int).

Note : La grammaire a été testée au CTI/ERM par un générateur d'automates (ref(22) de la bibliographie)

f. DESCRIPTION DE L'ALGORITHME SOUS FORME D'UNE GRAMMAIRE GENERATIVE

La grammaire se compose d'un ensemble metasymboles, de terminaux et de non-terminaux qui sont décrits ci-dessous.

(1) Metasymboles

"."	: symbole de concaténation de deux éléments
"|"	: séparateur de deux alternatives
"+"	: suivi d'un terminal (T) ou d'un non-terminal (NT) veut dire : au moins une occurence
"*"	: suivi d'un T ou NT veut dire : zéro ou plusieurs occurences
";"	: fin de production
"[" et "]"	: entourant un NT veut dire : zéro ou une occurence
"(" et ")"	: entourant des T et des NT chargent les règles de priorité
"<" et ">"	: entourent un symbole et le définissent comme NT
"::="	: symbole de production, veut dire qu'on peut remplacer les NT du membre de gauche par une des alternatives du membre de droite.

(2) Non-terminaux (NT)

Les non-terminaux constituent des noeuds de l'arbre grammatical

engendré par la grammaire. Un NT est le sommet de l'arbre ou d'un sous-arbre, qui coiffe toujours une chaîne de sortie correspondant au code généré pour effectuer la copie d'une valeur d'un mode déterminé.

Le tableau ci-dessous décrit, pour chaque NT, la valeur qui sera copiée par la chaîne de sortie coiffée par ce NT.

NT	*VALEUR*
< grammar >	arbre grammatical
< copy >	valeur source de tout mode
< dyn >	valeur ayant - une partie dynamique ou bien - une partie ayant une portée - une portée
< union >	valeur de mode union (...,...)
< struct >	valeur de mode struct (...)
< rowof >	valeur de mode [] amode

(3) *Terminaux (T)*

Chaque terminal de la grammaire est souligné. Un terminal représente un ensemble d'instructions en code symbolique, qui fait une partie de la copie. Certains terminaux ne représentent que des commentaires dans la chaîne de sortie ; ces commentaires sont très utiles pour faire la mise au point des routines qui génèrent le code.
Le tableau ci-dessous décrit la fonction du code généré.

T	*CODE*
endfile	commentaire : fin de la copie de la valeur
stat	code qui permet de copier la partie statique de la valeur de adds vers addo.
scope	code qui permet de faire le test de portée
unbeg	a) commentaire : début du code qui copie

	une valeur de mode union b) Table de branchement vers le code de valeur du mode qui existe au moment de la copie.
unend	commentaire : fin du code de copie de valeur de mode union
strbeg	commentaire : début du code qui copie une valeur de mode struct
strend	commentaire : fin du code qui copie une valeur de mode struct
nil	commentaire : la partie statique de la valeur a été copiée et il n'y a plus d'actions à faire pour cette valeur.
rowbeg	commentaire : début du code qui copie une valeur de mode [] amode
rowend	commentaire : fin du code qui copie une valeur de mode [] amode
statrow	code qui copie la partie statique des éléments d'une valeur multiple
loop	code qui permet le parcours des parties statiques des éléments d'une valeur multiple (pour aller chercher chaque fois sa partie dynamique).

(4) *Grammaire*

< grammar > ::= < copy >. endfile ;

< copy > ::= stat. [< dyn >] ;

< dyn > ::= scope | < union > | < struct > | < rowof >;

< union > ::= unbeg. (+ nil.< dyn > | < dyn >. nil | < dyn > . < dyn >). ⁂ (nil | < dyn >).
unend ;

< struct > ::= strbeg. ⁂ nil. < dyn >. ⁂ (nil | < dyn >). strend ;

< rowof > ::= rowbeg. statrow. [loop. < dyn >]. rowend ;

REMARQUE : Comme nous avons décrit en haut de la p13, on ne descend dans la production < union > que si au moins une des alternatives de la valeur de mode union correspondante possède une partie dynamique. Une valeur de mode union doit avoir au moins deux alternatives. Une valeur de ce mode, qui nous fait passer par la production < union >, a au

moins une des alternatives qui a une partie dynamique (pour au moins une des alternatives, on peut passer à la production < dyn >). On exprime cette idée en disant qu'une valeur de ce genre commence par

ou bien : + nil . < dyn > un certain nombre de nil (pas de partie dynamique) (au moins un) suivi d'un < dyn >.

ou bien : < dyn >. nil un <dyn > suivi d'un nil.

ou bien : < dyn >. <dyn> deux <dyn> qui se suivent.

les autres alternatives peuvent être n'importe quoi :

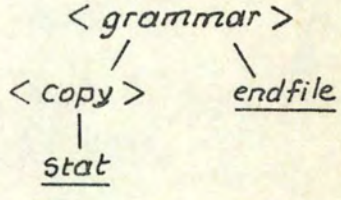
* (nil | < dyn >) un certain nombre de fois nil ou < dyn >.

Pour la production < struct > il y a le même raisonnement, mais là, il ne faut qu'un champ.

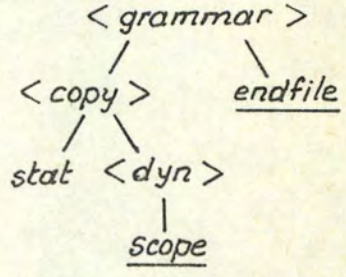
(5) *Exemples*

Pour chaque exemple on donne l'arbre grammatical avec la chaîne de sortie. Pour chaque valeur à copier on donne le mode correspondant d'une façon simplifiée.

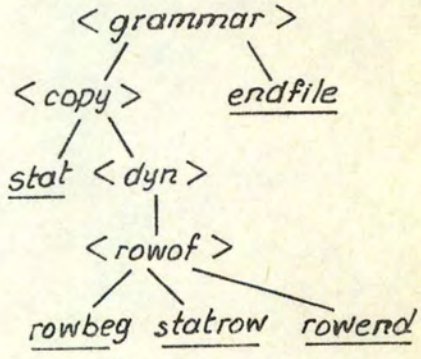
(a) struct (int, real) ;



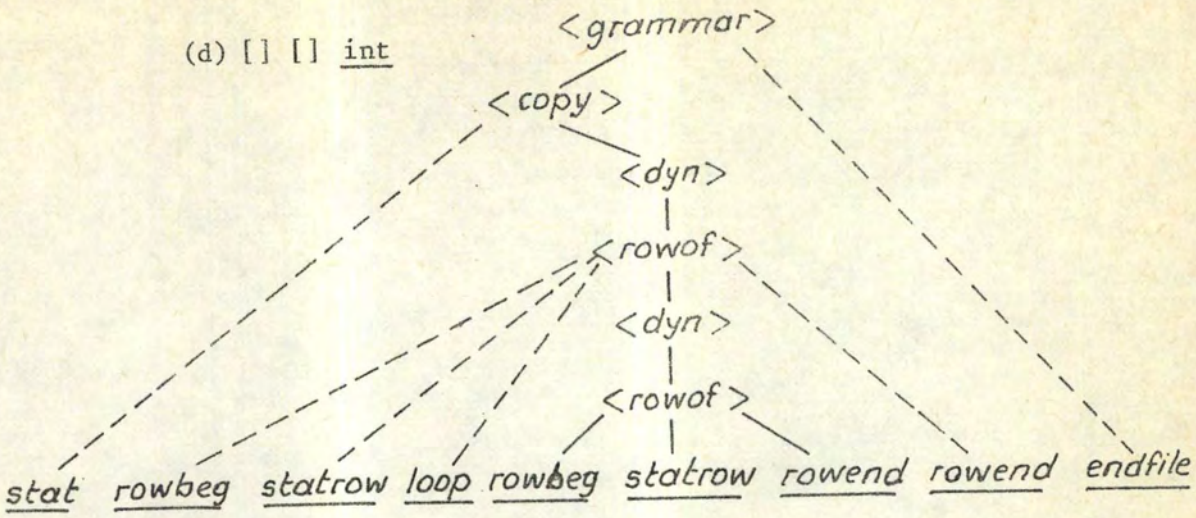
(b) union (int, ref real) ;



(c) [] real



(d) [] [] int


<grammar>
<copy>
<dyn>
<rowof>
<dyn>
<rowof>
stat
rowbeg
statrow
loop
rowbeg
statrow
rowend
rowend
endfile

CHAPITRE 2 : APPROCHE PRATIQUE DU PROBLEME

Jusqu'ici on a vu d'une façon fondamentale comment générer le code optimal qui copie une valeur ALGOL68 pendant l'exécution du programme. Les notions que nous avons vues sont complètement indépendantes de la machine sur laquelle le module de génération devra fonctionner. On n'a pas encore tenu compte de l'organisation physique de la mémoire pendant le déroulement d'un programme ALGOL68.

Il ne nous est pas possible de faire une implémentation complète de tous les accessoires "software" qu'on aurait besoin pour le bon fonctionnement du module et ensuite de l'exécution du code généré. Cependant, il nous est possible de décrire une organisation réalisable de la mémoire. Cette organisation pour l'instant reste fictive. Une fois cette description faite il nous est facile de simuler certaines parties nécessaire au fonctionnement du module (en même temps on en testera le bon déroulement).

Dans ce qui suit on décrit successivement . l'organisation globale de la mémoire pendant l'exécution d'un programme, la représentation détaillée des valeurs et l'organisation détaillée de la mémoire avec l'implémentation des valeurs.

a. ORGANISATION DE LA MEMOIRE PENDANT L'EXECUTION D'UN PROGRAMME ALGOL68

ALGOL68 est un langage qui a une structure de bloc. Cette structure de bloc est telle que certaines valeurs ne vivent que la durée d'un bloc. Il serait interessant de pouvoir éliminer les valeurs de la mémoire appartenant à des blocs qui ne sont pas en vie. On ne créerait la valeur que physiquement lorsque pendant le déroulement du programme on entre dans un bloc. Illustrons cette notion de bloc par l'exemple suivant :

```
prog1 : ① begin int i ;
        ② begin int j ;
          end
        ③ begin int k ;
          .
          .
          .
          end
      end
```

la valeur i reste en mémoire pendant la vie du bloc ①

En sortant du bloc ② on récupère la place que prennent les valeurs déclarées dans ce bloc (comme j). En entrant dans ③ on peut mettre la valeur k à l'emplacement où se trouvaient les valeurs du bloc ②.

Afin de garder le principe du bloc d'une façon efficace, il serait facile d'organiser la mémoire de telle sorte qu'on puisse y ajouter des valeurs et en retirer sans grande difficulté. Un pile répond complètement à ce principe. Une pile est une zone de mémoire qui s'accroît d'un côté, (disons le sommet). On ne peut ajouter une partie de mémoire à cette pile que par le sommet et on peut en enlever que par la même voie. Si on reprend l'exemple ci-dessus, on peut, en entrant dans le bloc ① placer la valeur de i au sommet de la pile puis en entrant en deux, on place la valeur de j au nouveau sommet ; en sortant de ② on libère la place prise par j qu'est rendue disponible à la valeur de k dont on aura besoin en entrant en ③ .

(1) Notion de procédure vue sous l'angle du bloc

Un des modes en ALGOL68 est le mode proc. Une procédure est un morceau de programme qui peut être appelé pendant l'exécution à un autre endroit du programme que celui ou il est décrit.

Une procédure peut avoir plusieurs blocs intérieurs. Pendant son exécution, une procédure équivaut à un bloc, où l'on entre par le début.

L'exemple suivant nous montre le bloc qui est exécuté à l'appel d'une procédure.

```
prog2 : begin int j : = 10 ;
              proc a = (int i) void :
                       begin real x ;

                       end;

        a (j)
        end
```

A l'appel a (j) la procédure est remplacée par le bloc suivant :

```
        begin int i = j ;
          begin real x ;

          end
        end
```

(2) *Organisation globale de la mémoire pendant l'exécution*

Il existe deux stratégies pour la gestion de la mémoire utilisée comme pile. On peut allouer une zone de données (au sommet de la pile) à chaque entrée de bloc, en la désallouant à chaque sortie. On peut n'allouer une zone qu'à chaque appel de procédure. Chaque méthode a ses avantages. La première nous évite toute perte de place, car on ne garde aucune valeur d'un bloc qui n'est pas en vie. Mais elle demande beaucoup d'actions de création et de suppression de zones de données. La deuxième méthode nous fait perdre un peu de place. En entrant dans la procédure, ou crée une zone de données qui peut tenir toutes les valeurs pouvant être en vie en même temps. La plupart du temps, toutes ces valeurs ne sont pas en vie, en même temps, d'où perte de place. On évite la surabondance de création et de suppression de zones de données.

Pendant le déroulement d'un programme, il n'y a pas seulement une zone de données qui se trouve en mémoire, il y a encore autre chose. Essentiellement la mémoire est partagée en trois parties.

(a) Zone programme (ZP)

Cette zone contient tous les éléments qui ont une adresse fixe pendant le déroulement du programme. On y trouve le code généré, une table des modes etc ....

(b) Zone des données qui suivent le mécanisme de la pile (RANST)

On y trouve toutes les valeurs qui sont déclarées dans des blocs et qui ont comme durée de vie, celle du bloc.

(c) Zone des données ne suivant pas le mécanisme de la pile (HEAP)

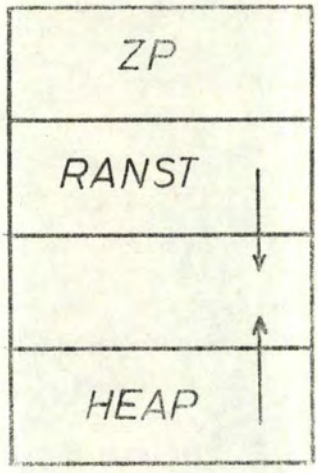
On y trouve toutes les valeurs qui ne peuvent être gardées dans la zone RANST

(d) Remarque

La taille de ZP est fixe, une fois que le programme se déroule. La taille de RANST et de HEAP sont variables. Chacune de ces deux zones de données suivent le mécanisme de la pile.
Le début de (ZP) est la plus petite adresse disponible pour le programme. Cette zone est suivie immédiatement par la pile RANST qui croît selon les valeurs ascendantes des adresses. La pile

HEAP commence à la plus grande adresse alouée à notre programme et croît selon les valeurs décroissantes des adresses.
On dit que la mémoire est pleine quand il y a collision entre la pile RANST et la pile HEAP

Représentation schématique de la mémoire



b. REPRESENTATION DETAILLEE DES VALEURS ALGOL68

La représentation d'une valeur peut comporter différents emplacements élémentaires de la mémoire qui ne sont pas nécessairement consécutifs. On décrit ici le principe du stockage des valeurs. Ceci nous aidera à les implémenter plus tard. La description des valeurs est basée sur la définition récursive de partie statique et de partie dynamique d'une valeur. Pour chaque valeur, la partie statique et la partie dynamique sont telles que la façon de stocker la première peut être déduite du mode de la valeur ; ce n'est pas le cas pour la dernière le stockage de celle-ci dépend du déroulement du programme.

Comme exemple : la taille du descripteur d'une valeur multiple et le découpage de ce descripteur ne dépendent que du mode de la valeur (nombre de dimensions). Ce mode est spécifié par la déclaration" [l_1:u_1, ..., l_n : u_n] amode". On dit que le descripteur est la partie statique de la valeur. Par contre, le nombre d'éléments qui constituent la partie dynamique dépend de l'élaboration des bornes l_1, u_1, ..., l_n , u_n dans le programme.

Dans ce texte, les termes statique et dynamique, utilisés pour n'importe quelle propriété sont équivalents à, respectivement, déductible du mode et dépendant de l'élaboration.

(1) Remarque sur l'accès à une valeur

Il faut pouvoir accéder, à tout instant, à chaque valeur en vie pendant le déroulement du programme. Cet accès à une valeur doit être efficace. On dit que l'accès est une propriété statique d'une valeur. On connait statiquement le bloc dans lequel une valeur a été déclarée, il s'en suit que la valeur est en vie, si la procédure dans laquelle le bloc existe, est activée. Au sein de cette procédure chaque bloc normal a une position relative qui est inchangeable. Au sein de chaque bloc simple, une valeur a aussi une position fixe. La profondeur d'une procédure est aussi connue statiquement. Il s'en suit, qu'on peut définir l'accès à une valeur par le couple (n,p) ou n = profondeur de la procédure

p = position relative de la valeur dans cette procédure.

Exemple :

```
prog3 : begin int i1 ;
            real x1 ;
            proc proc1 =
                 void : begin int i2 ;
                             real x2 ;
                             .
                             .
                             .
                        end ;
            .
            .
            .
        end
```

Le programme principal a une profondeur o et la procédure proc1 a une profondeur 1. Les valeurs correspondant aux variables de l'exemple peuvent être définies comme suit :

Valeur	*couple (n,p)*
i1	0,1
x1	0,2
proc1	0,3
i2	1,1
x2	1,2

On verra plus tard, qu'à partir de ces couples, et à partir du mode donnant la taille de la partie statique, on peut retrouver physiquement chaque valeur dans la mémoire.

(2) *Représentation mémoire des valeurs*

On donne ci-dessous la représentation mémoire des valeurs des modes qu'on a décrit en 1.b (p 4). Comme unité de longueur, on a pris le caractère. La taille de la partie statique de chaque valeur sera un certain nombre de caractères.

(a) Valeurs de modes de base

Ces valeurs n'ont qu'une partie statique. Leur taille dans notre implémentation est donnée par le tableau suivant.

Valeur de mode	*taille*
int	2
real	4
bool	1
char	1
bits	1
bytes	1

Les valeurs des modes int, real, bits et bytes peuvent être de modes long. Pour la valeur de mode long .... long amode (n fois long) la taille sera de n fois la taille d'une valeur de base de ce mode.

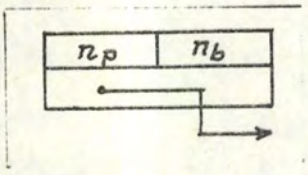
Exemple = real x = 3.14 | 3.14 | taille = 4

long real xx = 3.14 | 3.14 | taille = 8

(b) Valeur de type ref amode

Ces valeurs n'ont qu'une partie statique. Comme ces valeurs ont une portée, il faudra garder la représentation de cette portée dans la partie statique (définition de portée 1C p 6). On a vu qu'on peut dire que la portée est un bloc. Une valeur est représentée statiquement par un couple (n,p). Ce couple suffit pour retrouver la valeur pendant l'exécution. Mais ce couple ne suffit pas pour savoir dans quel bloc se trouve la valeur (on ne connait que la procédure (n)). La portée correspond à un bloc bien déterminé dans la procédure, ce bloc peut être caractérisé par son numéro de profondeur dans cette procédure. Le couple (np, nb) (numéro procédure et numéro bloc dans la procédure) localisant parfaitement un bloc, peut servir de couple qui caractérise la portée d'une valeur.

Une valeur de mode ref amode est représentée comme suit :



portée

pointeur vers la valeur qui en est la référence.

taille : np : 2
nb : 2
pointeur : 4
taille totale : 8

Exemple

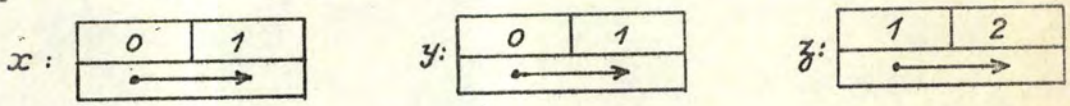
```
prog4 : begin int x ; real y;                    [1]  (1)
            proc a = (int i) void :              [2]
               begin                                  (1)
                     begin real z;                    (2)
                     end
               end   ;
```

numéro's de procédures [i]
numéro's de blocs (j)

Remarque :

Les numéros de bloc semblent commencer par 1, ceci est dû au fait que le niveau zéro est réservé au bloc extérieur qui est crée au moment de l'appel. On y retrouve les paramètres de la procédure.

Représentation des valeurs x, y, z,



(c) Valeurs de mode proc

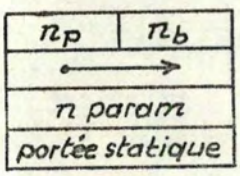
Une valeur de mode proc est en fait une routine qui a comme portée le plus petit des portées des valeurs employées dans la procédure, les variables locales exceptées. Il va de soi qu'on ne mettra pas le texte de la routine dans la zone des données. On pourrait donc représenter la valeur comme le couple (portée, pointeur) le pointeur pointe vers de la routine). Pour des raisons de facilité de gestion de la mémoire qui ne seront pas approfondies ici, on a ajouté deux champs à la valeur. Un premier champ est nparam et le deuxième est appelé portée statique.

nparam:représente le déplacement par rapport au début de la zone des données, le début de la zone qui contient les paramètres de la procédure. On a vu que cette zone correspond au bloc de niveau zéro de cette procédure.
On garde nparam dans la valeur pour des raisons d'implémentation à l'ERM. Il est facile de connaître à l'appel de procédure le début de la zone des paramètres.

portée statique : chaque procédure a une portée statique, c'est le couple (np,nb) (numéro procédure, numéro bloc) dans lequel la procédure a été déclarée. Dans le champ portée statique on garde un pointeur vers le début de la zone des données de la procédure qui contient le bloc qui est la portée statique de cette procédure. Ceci est instauré pour des raisons d'implémentation à l'ERM.
A chaque appel de procédure il faut pouvoir accéder à tou-

tes les variables de la procédure de portée statique, en gardant le couple (np,nb) dans la valeur on peut à l'appel de procédure calculer l'adresse machine correspondante au début de la zone de données correspondante. Dans notre implémentation le régistre général n°10 contient toujours cette adresse. Il sera garni facilement grâce à la portée statique.

Une valeur de mode proc se représente alors comme suit :

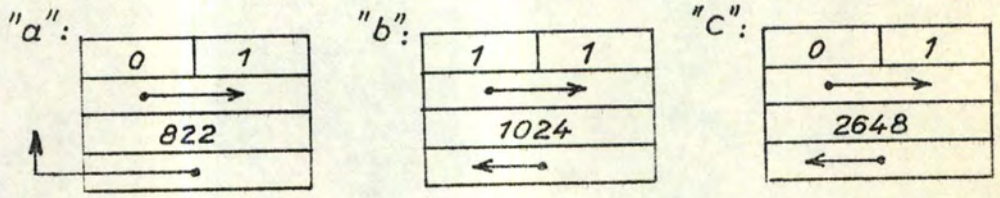


taille np et nb : chacun : 2
les autres champs : 4
taille totale : 16

Exemple : dans cet exemple on prend les mêmes conventions que dans celui des valeurs de ref amode

```
prog5 : begin real x;                                    [o]   (1)
          proc a = void :                                [1]
            begin real y;                                      (2)
              proc b = (int i) ref int :                 [2]
                begin                                          (1)
                    y:=x;
                end
              proc c = void :                                  (2)
                begin                                          (1)
                    x:=2;
                end;
        end
      end
```

Représentation des valeurs de a, b, c,



(d) Valeurs de mode [,, ...,] amode

Une valeur de mode [,,...,] amode comporte une partie statique le descripteur (𝒟), et une patie dynamique (les éléments de la valeur multiple). Le descripteur contient l'information nécessaire au calcul de l'adresse d'un élément quelconque de la valeur. Cette information est dynamique. Dans notre implémentation le descripteur comprend :

pointeur : adresse du premier élément de la valeur

flag : élément qui donne une information sur la consécutivité des éléments de la partie dynamique.

flag = 1 : éléments de la partie dynamique sont consécutifs en mémoire.

flag = 2 : éléments de la partie dynamique ne sont pas consécutifs en mémoire.

d_o taille total de la partie statique des éléments

Par dimension de la valeur multiple on trouve les trois champs suivants :

l_i : borne inférieure correspondant à la dimension i

u_i : borne supérieure correspondant à la dimension i

d_i : taille totale d'un élément de la dimension i. Expliquons la signification de "d_i" par un exemple. Soit la valeur multiple x à n dimensions, déclarée $[l_1 : u_1, \ldots . l_n : u_n]$ amode x

La taille d'un élément correspondant à la dimension i est la taille d'une sous-valeur qui est définie comme suit :
$x[k_1, \ldots\ldots, k_{i-1}, k_i, l_{i+1} : u_{i+1}, \ldots., l_n : u_n]$ ou les k_j sont des valeurs entières avec $lj \leqslant kj \leqslant uj$. $j \in [1,i]$.
Toutes les sous-valeurs correspondantes à la dimension i sont

$x[k_1, \ldots., k_{i-1}, l_i, l_{i+1} : u_{i+1}, \ldots, l_n : u_n]$
$x[k_1, \ldots., k_{i-1}, l_{i+1}, l_{i+1} : u_{i+1}, \ldots, l_n : u_n]$
$\vdots$
$x[k_1, \ldots., k_{i-1}, u_i, l_{i+1} : u_{i+1}, \ldots, l_n : u_n]$

Il y en a $t_i = u_i - l_i + 1$

Les cas limites de la valeur d_i sont d_o qui est décrite plus haut, et d_n qui correspond à une sous-valeur $x[k_1, \ldots, k_n]$ qui est un élément de base de la valeur multiple. La taille de cet é-

lément est connu statiquement, le mode des éléments d'une valeur multiple étant connu par la déclaration.

On peut exprimer la fonction qui calcule d_i pour chaque dimension, la taille d'un élément d'une dimension est la somme des tailles de tous les éléments de la dimension supérieure. Ce qui s'exprime sous forme analytique :

(a) Les éléments sont consécutifs en mémoire

d_n = taille d'un élément

$d_i = d_{i+1} * t_{i+1}$ pour $i \in [n-1,0]$

(b) Les éléments ne sont pas consécutifs en mémoire

h_i : taille d'un trou laissé entre deux éléments de dimension i.

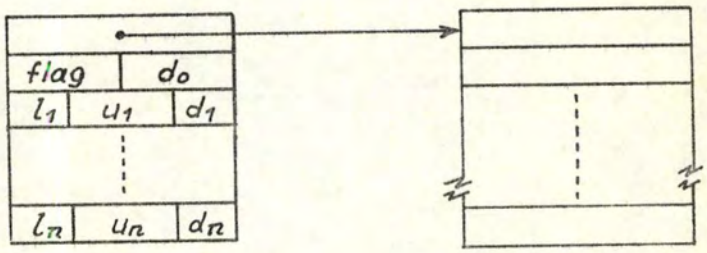
d_n : taille d'un élément + h_n

$d_i : d_{i+1} * t_{i+1} + h_i$ pour $i \in [n-1,1]$

$d_o : d_1 * t_1 - \sum_{i=1}^{n} h_i$

Une valeur de mode [,,...,] amode se représente comme suit :

Partie statique — *Descripteur D* | *Partie dynamique* — *éléments*



taille pointeur : 4

flag : 2

autres éléments : 2

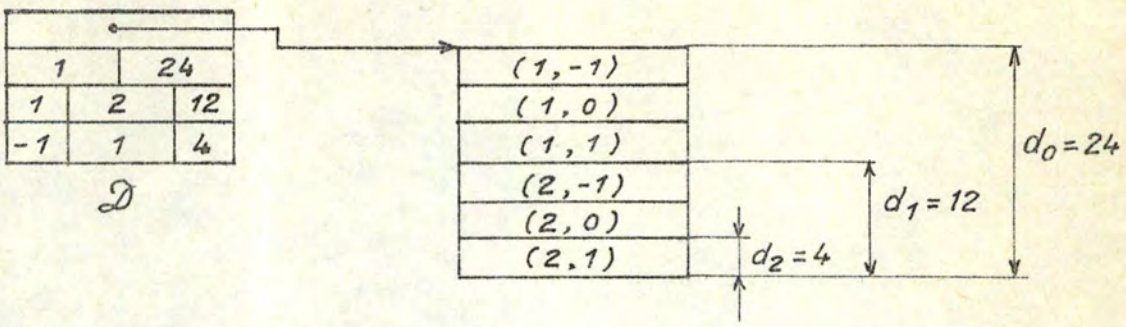
taille totale : 6n + 8

Exemple :

on donne une valeur multiple à deux dimensions,

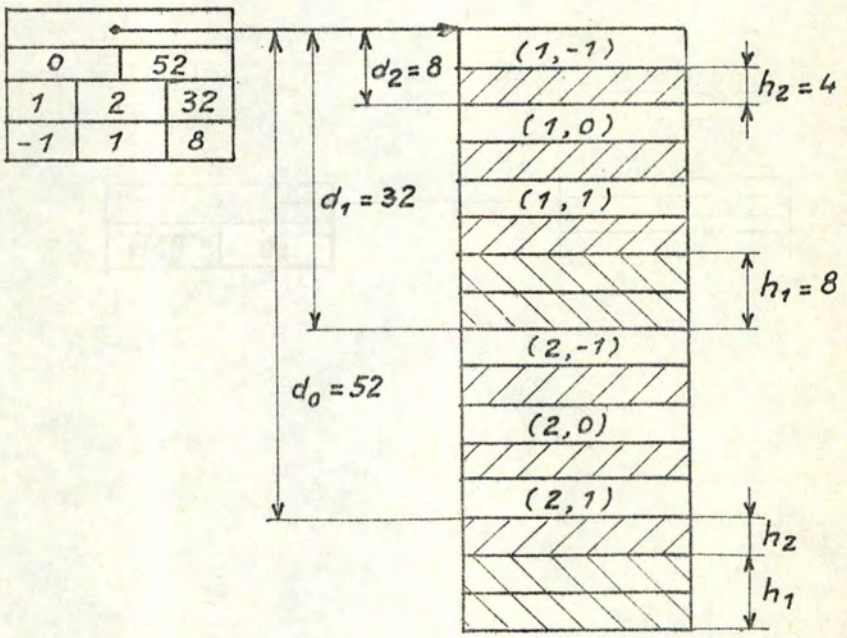
[1 : 2, - 1 : 1] real x = (.....) ;

(i) image de la valeur si les éléments sont consécutifs en mémoire.



(ii) image de la valeur si les éléments ne sont pas consécutifs en mémoire.

Partie statique *Partie dynamique (les zones hachurées sont les trous)*



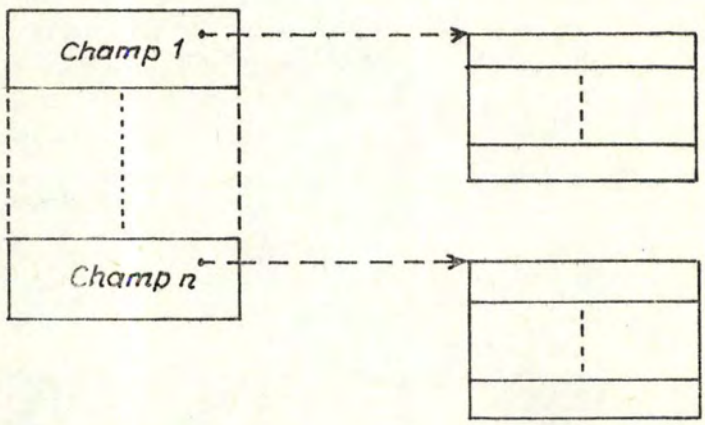
(e) Valeurs de mode struct (modea a, ..., modez z)

Une valeur de ce mode se compose d'une partie statique, qui est l'ensemble des parties statiques de tous ses champs, et d'une partie dynamique qui est l'ensemble des parties dynamiques de

tous les champs. Pour que l'adressage de chaque champ de la valeur puisse se faire d'une façon efficace il est nécessaire que cela puisse être fait statiquement. Pour ce faire on met la partie statique de tous les champs consécutivement en mémoire.

Une valeur de mode struct (modea a, ..., modez z) se représente donc comme suit : les flèches pointillées n'existent que si le champ à une partie dynamique.

Partie statique *Partie dynamique*

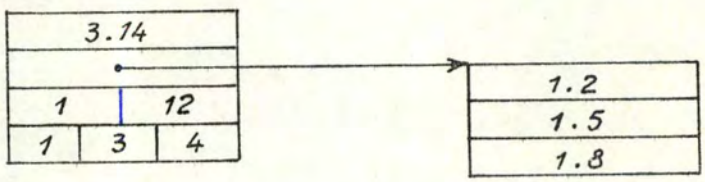


taille = ∑tailles statiques des différents champs

Exemple

struct (real x, [1:3]real y) z = (3.14, (1.2, 1.5, 1.8)) ;

Partie statique *Partie dynamique*

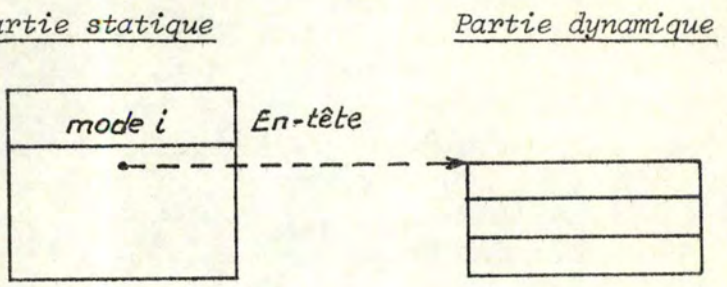


(f) Valeurs qui sont de mode union (modea,..., modez)

Une valeur de ce mode se compose d'un en-tête, dans lequel on place dynamiquement une information du mode en cours et d'une

place mémoire qui est d'une taille telle qu'elle peut contenir la partie statique la plus grande de toutes les parties statiques décrites dans l'union.

Représentation d'une valeur de mode union (modea,...., modez)

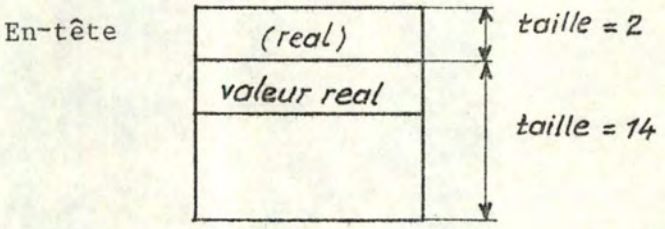


taille : en-tête:2

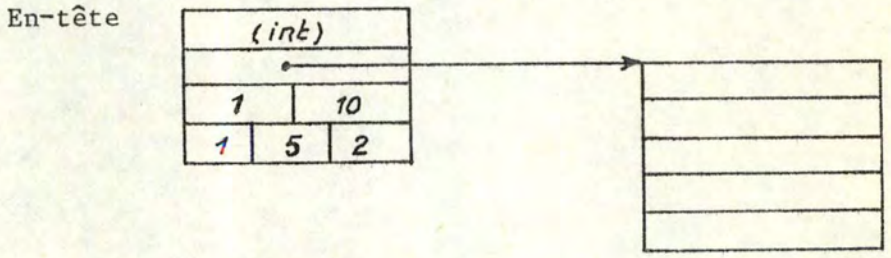
reste:taille statique maximale de modei, i∈[a,..., z]

Exemple : valeur de mode union (real, [1:5]int) x

Si le mode actuel est real on a :



Si le mode actuel est [1:5]int ou a



(g) Accès définitif à la valeur

On a vu plus haut que l'accès à une valeur peut être représenté par un couple (n,p). Nous ne modifierons pas ce que nous

avons dit en ce qui concerne "n", (n reste le numéro de procédure). Maintenant qu'on a décrit l'image des valeurs en mémoire et leur taille statique on peut décrire définitivement p, "p" est le déplacement de la valeur par rapport au début de la zone des données de la procédure ou cette valeur a été déclarée. C'est une valeur statique, car la position relative d'une valeur est connue dans la zone des données et on connait aussi la taille de la partie statique de toutes les valeurs qui précèdent notre valeur en mémoire. D'ou l'on peut facilement calculer ce déplacement.

c. ORGANISATION DETAILLEE DE LA MEMOIRE ET IMPLEMENTATION DES VALEURS

On a étudié la configuration globale de la mémoire (en 2a), une étude plus approfondie sera faite de la partie de la mémoire RANST pendant le déroulement d'un programme. L'organisation des parties ZP et HEAP n'apportent rien à la compréhention du travail fait à l'occasion de ce mémoire.

(1) Organisation conceptuelle de RANST

La mémoire RANST est exploitée comme une pile. Cela veut dire qu'à chaque entrée de procédure on crée une zone de données pour cette procédure. Cette zone de données se trouve toujours au sommet de la pile. En sortant de la procédure on enlève la zone de données du sommet de la pile. Cette place libérée est rendue disponible dès ce moment. A chaque instant du déroulement du programme, un registre pointe vers le sommet de cette pile. Nommons ce régistre "rnstpm".

A chaque instant on trouve aussi un pointeur qui pointe vers le début de la "DA" qui se trouve au sommet. C'est la "DA" de la procédure en cours d'exécution. Au moment ou on enlève une "DA" de la pile il est indispensable de remettre ce pointeur à jour (on le nomme "topda"). On doit aussi remettre "rnstpm" à jour à ce moment là. Pour ce faire on a instauré un chaînage entre les différentes "DA". Pour des raisons de conventions IBM dont nous devons tenir compte, le chaînage est fait dans les deux sens, de haut en bas et de bas en haut.

En plus du chaînage décrit ci-dessus, on en trouve encore un autre. Une procédure peut être appelée à n'importe quel instant dans le

---

note : la notation "DA" provient de "data area",
c'est la zone des données.

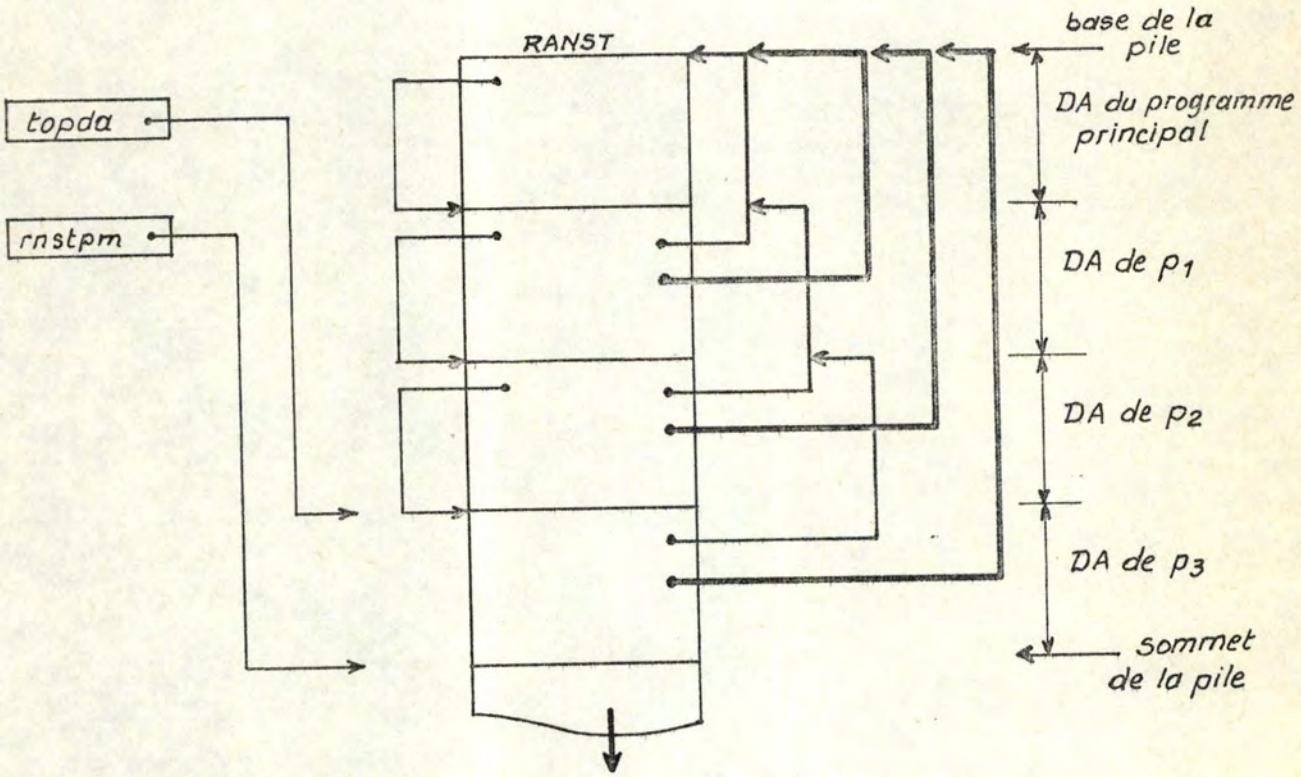
programme. Comme il est nécessaire que l'on puisse accéder à partir de la "DA" de cette procédure, à la "DA" de la procédure dans laquelle la première a été déclarée, on fera aussi ce chaînage.

On appelle ce dernier chaînage le chaînage statique et le premier est appelé chaînage dynamique.

En employant le programme ci-dessous, on peut montrer la configuration conceptuelle de la pile RANST à l'appel de la procédure p .

```
prog6 : begin proc p1 = void : begin p2 ;  end ;
                proc p2 = void : begin p3 ;  end ;
                proc p3 = void : begin  -    end ;

                p1;

        end
```

Configuration de RANST



Rem : chaîne statique :

chaîne dynamique :

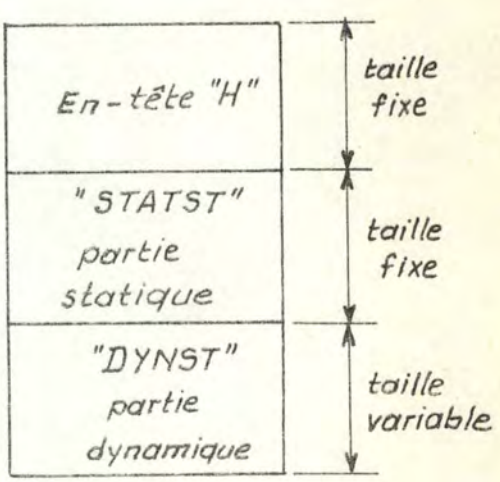
(2) *Etude d'une zone de données (DA) d'une procédure*

Une zone de données est subdivisée en trois parties. On y retrouve d'abord un en-tête, dans laquelle on trouve toutes les informations d'allure générale. Ensuite une zone où seront mises toutes les parties statiques des valeurs appartenant à la procédure, et, en dernier lieu, on a la zone ou on retrouve toutes les parties dynamiques des valeurs de la procédure. Cette zone dynamique est subdivisée en autant de parties qu'il y a de blocs dans la procédure plus la zone dynamique des paramètres.

On peut se demander pourquoi, quand on crée une "DA" par procédure et que l'accès aux valeurs est connu par rapport au début de la procédure, on va subdiviser les parties dynamiques par blocs. Ceci est fait pour récupérer le maximum de place. On a dit précédemment qu'on gardait sur la pile dans la "DA" de la procédure toutes les valeurs de cette procédure, cela est partiellement faux. On ne garde dans cette "DA" que la partie statique de ces valeurs et s'il existe plusieurs blocs dans une procédure qui ont un même numéro de profondeur, cette partie statique de valeurs se trouve à la même place. Il n'y a qu'un bloc d'une certaine profondeur en vie à la fois. La partie dynamique des valeurs appartenant à un bloc qui n'est plus en vie est enlevé du sommet de la pile. Comme en général c'est la partie dynamique d'une valeur qui prend le plus de place, on peut dire que la perte de place due aux parties statiques et minime. (ou étudiera ce problème plus loin).

Configuration de la "DA" correspondant au programme suivant :

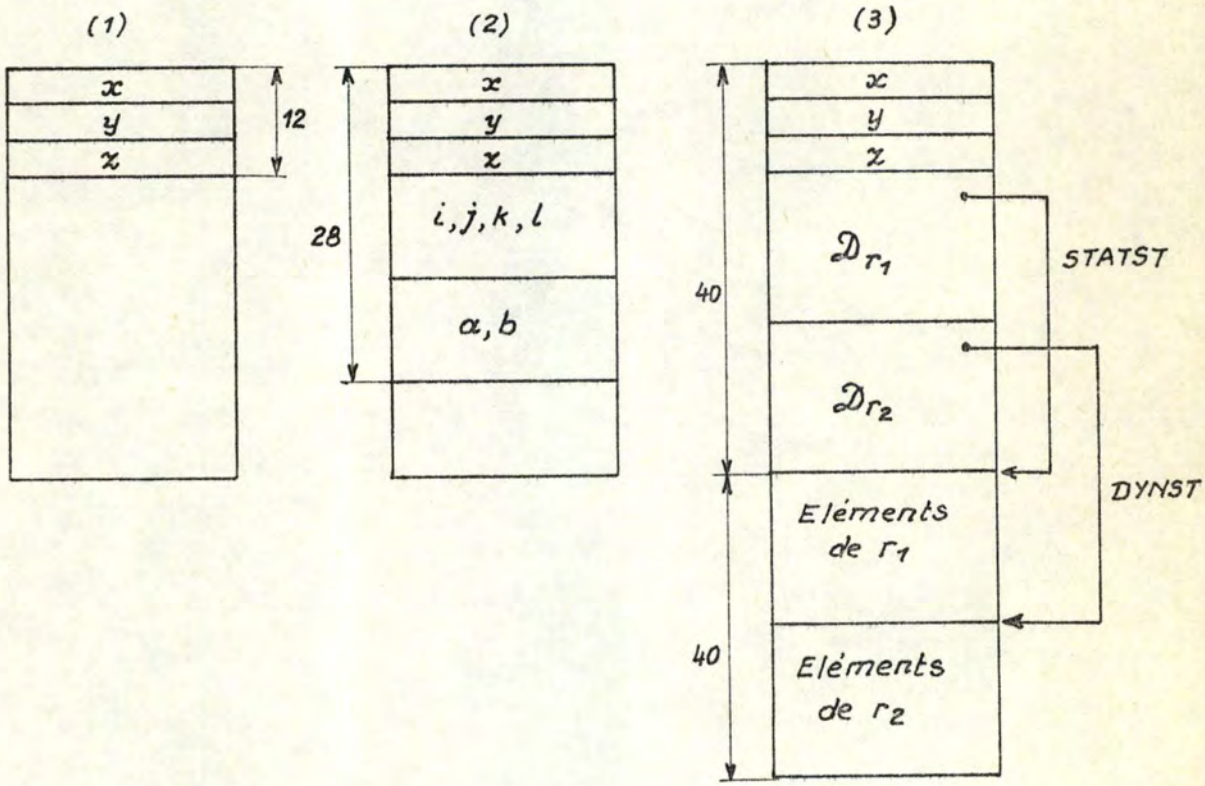
```
prog7 : begin real x, y, z ;          (1)
          begin int   j, k, l ;       (2)
                real a, b ;

        end;
        begin [ 1:5] real r1,r2 ;     (2)

        end

end
```



(3) *Etude de la partie STATST et DYNST d'une zone de données*

L'emplacement de chaque valeur dans la partie STATST est statique. Au fond, on ne retrouve dans cette partie que la partie statique des valeurs des blocs encore en vie. Les autres valeurs pouvant être surimprimées. La taille de la partie STATST sera la taille maximale qui sera requise durant la vie de la procédure. Cette taille peut être calculée statiquement.

L'entrée et la sortie d'un bloc au sein d'une procédure sont connues statiquement dans le programme. Il nous sera dès lors possible de prévoir la création et la suppression d'une zone de DYNST à chaque entrée et sortie de bloc (ces mécanismes ne seront pas décrits dans ce mémoire) pendant le déroulement du programme. Illustrons ces principes en utilisant l'exemple prog7. On verra STATST et DYNST à trois moments de l'exécution. A l'entrée du premier bloc, puis à l'entrée du deuxième bloc et enfin à l'entrée du troisième bloc, chaque fois après l'élaboration des déclarations.



(4) Implémentation des valeurs dans la "DA"

On a vu en (1 d p8) qu'il est nécessaire d'implémenter les valeurs d'une certaine façon pour diminuer les problèmes de surimpression. Il faudra respecter certaines règles.

Règle 1 : On ne trouve jamais de partie dynamique d'une valeur dans STATST

Règle 2 : Toute la partie dynamique sera placée dans DYNST ou bien sur HEAP.

Règle 3 : La partie statique d'une valeur sera toujours placée en cases consécutives de la mémoire.

Règle 4 : Tous les pointeurs internes d'une valeur sur RANST doivent avoir la même direction (vers le sommet de RANST).

(a) Description conceptuelle de l'implémentation d'une valeur sur RANST

On place d'abord la partie statique de la valeur à un emplacement déterminé statiquement (n,p) de STATST. Ensuite la partie dynamique de la valeur, si elle existe, est placée au sommet de DYNST.

Cette partie dynamique se place séquentiellement et récussivement à ce sommet. On place d'abord en bloc toutes les parties statiques suivies ensuite des parties dynamiques correspondantes de la partie dynamique.

(b) Description formelle de l'implémentation d'une valeur sur RANST

La description de l'algorithme est faite sous forme de grammaire générative. La chaîne de sortie de cette grammaire décrit la séquence des éléments retrouvés sur RANST.

(i) Métasymboles : ce sont les mêmes que ceux utilisées en 1f(1) p13.

(ii) Non-terminaux: Les NT <union> , <struct> et <rowof> ont la même signification qu'en 1f(2) p14. On décrit seulement <val> et <dyn> .

<val> : sommet de l'arbre grammatical

<dyn> : noeud de l'arbre correspondant a une valeur ayant une partie dynamique.

(iii) Terminaux (T)

T	*SIGNIFICATION*
stats	partie statique de la valeur (placée sur STATST)
statd	partie statique d'un élément (placée sur DYNST)
ub	commentaire : début de valeur de mode union
ue	commentaire : fin de valeur de mode union
sb	commentaire : début de valeur de mode struct
se	commentaire : fin de valeur de mode struct
rb	commentaire : début de valeur multiple
re	commentaire : fin de valeur multiple
loopi	quand on trouve loopi dans la chaîne de sortie, on sait qu'on est passé par la production 5. On place donc les éléments d'une valeur multiple sur DYNST. Ces éléments sont encadrés par rb et re. Le commentaire loopi veut dire que tout ce qu'on trouve dans la chaîne de sortie entre loopi et le re correspondant sera placé autant de fois sur DYNST qu'il y a d'éléments dans la valeur multiple.
nil	commentaire : rien n'est placé sur DYNST

(iv) Grammaire

(1) <val> :: = stats.[<dyn>] ;

(2) <dyn> :: = <union>|<struct>|<rowof>;

(3) <union> :: = ub. (nil | <dyn>).+ (nil | <dyn>). ue ;

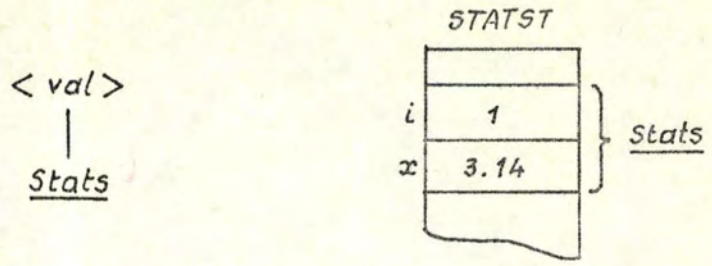
(4) <struct> :: = sb. + (nil | <dyn>). se ;

(5) <rowof> :: = rb. statd. [loopi.<dyn>] . re ;

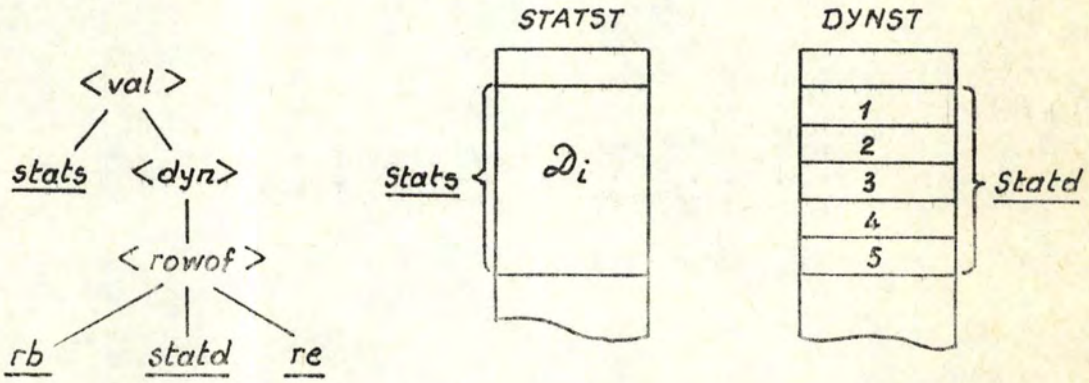
(v) Exemples :

Pour chaque exemple on donne l'arbre grammatical et la configuration de RANST pour la valeur traitée.

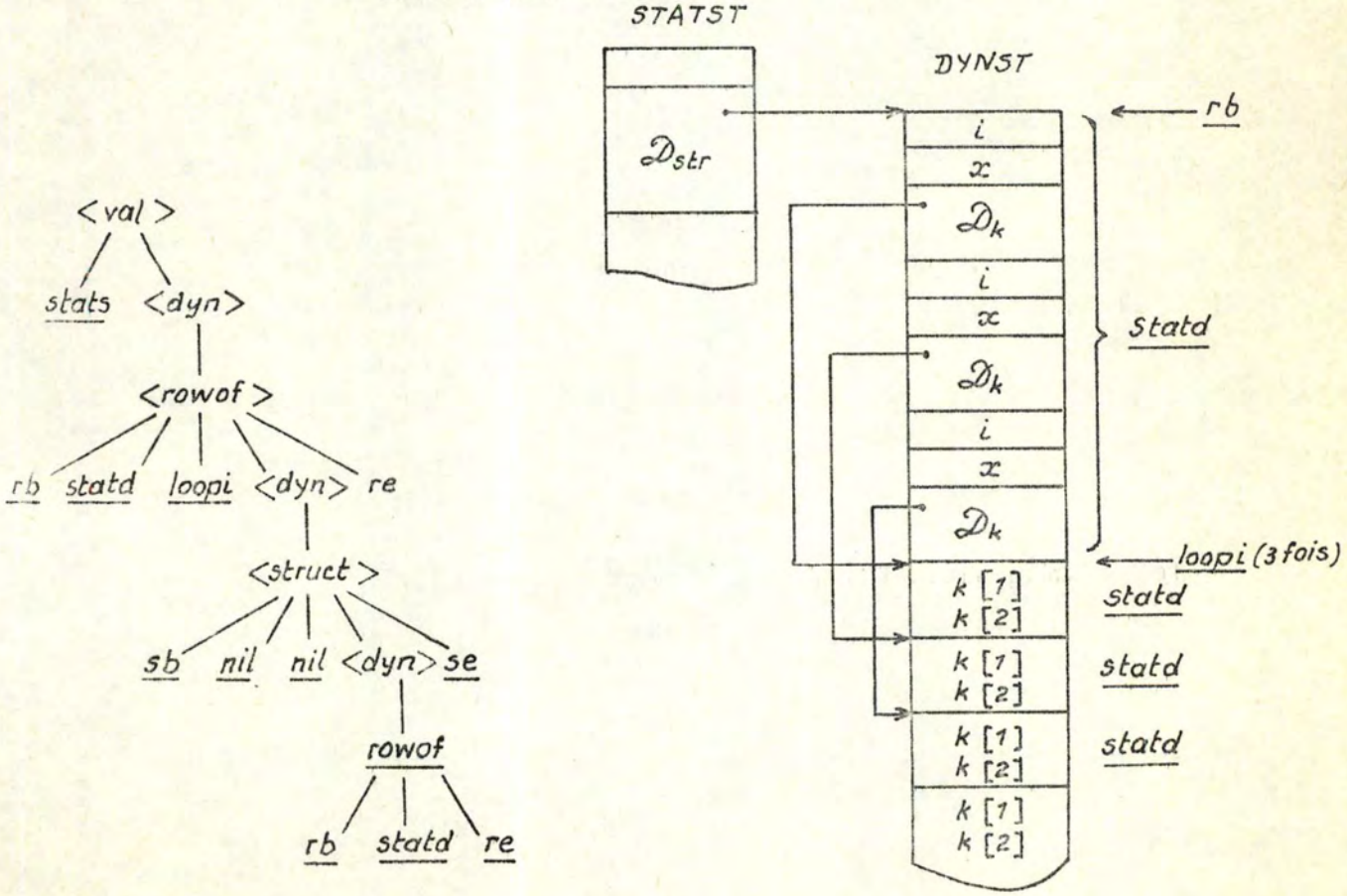
Struct (int i, real x) s = (1,3.14) ;



[1:5] int i = (1,2,3,4,5) ;



[1:3] struct (int i, real x, [1:2] int k) str = ... ;



(5) Etude de l'en-tête de la "DA"

L'en-tête de la "DA" d'une procédure sera essentiellement partagé en quatre parties. Unezone de sauvetage et trois zones de pointeurs.

(a) Zone de sauvetage : ("SAVEAREA")

Cette zone est la seule dans l'étude de la "DA" qui nous concerne immédiatement à la machine IBM. Sur ces machines, il existe des conventions en ce qui concerne les "DA" des procédures. La zone "SAVEAREA" est conforme aux conventions IBM. La zone a une longueur de 18 mots mémoire ou chaque mot a sa signification. On y retrouve les pointeurs de chaînage qu'on à évoqué en 2 C (1) p. 32 . On y trouve aussi l'emplacement de sauvegarde des régistres, il y en a 16. On décrira à dessous tous les champs de la zone "SAVEARA".

CHAMP	*DESCRIPTION*
1.	pointeur vers le sommet de STATST de la "DA"
2.	pointeur vers le début de la "DA" de la procédure appelante, c'est la chaîne dynamique de retour décrite en 2c (1) p 32 .
3.	pointeur vers la "DA" de la procédure appelée, c'est une chaîne dynamique aller.
4.	adresse de retour dans le code de la procédure appelante. C'est l'adresse de l'instruction à exécuter dès qu'on sort de la procédure en cours. C'est toujours le régistre 14 en IBM.
5.	adresse du début du code de la procédure en cours. C'est toujours le registre 15 en IBM.
6 - 15	contenu des registres 0 à 9
16	pointeur de chaînage statique décrit en 2c (1) p 32 c'est toujours le contenu du registre 10.
17 - 18	contenu des registres 11 - 12.

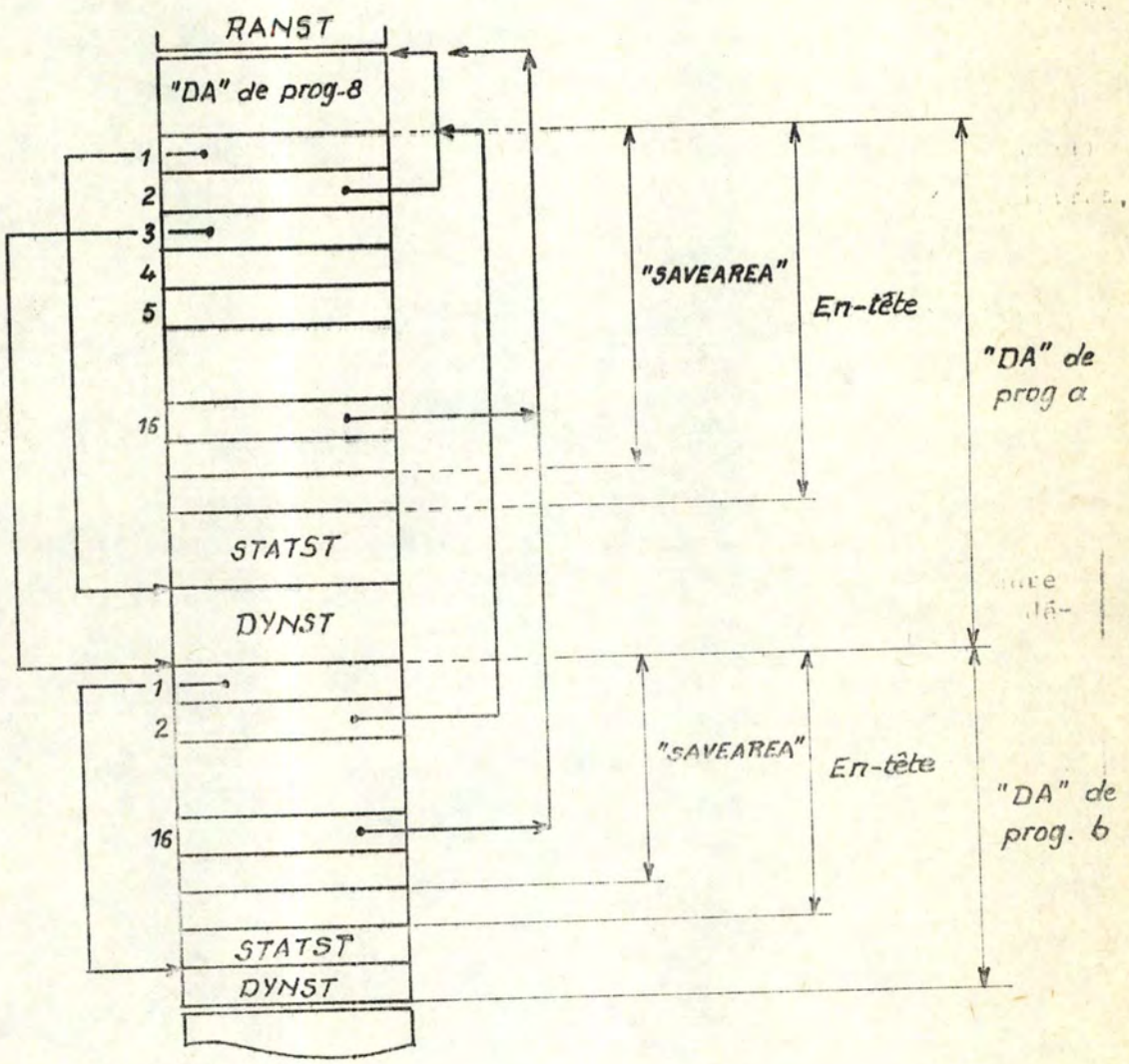
Remarque :

Le registre n°13 pointe toujours vers le début de la zone de données qui se trouve au sommet de la pile.

Exemple :

On représente ci-dessous la configuration de RANST et les différents champs de "SAVEAREA" pour le programme.

```
prog8 : begin proc a = void : begin .... b, .... end ;
                proc b = void : begin ........ end ;
                a ;
        end
```



(b) Première zone de pointeurs : "DISPLAY"

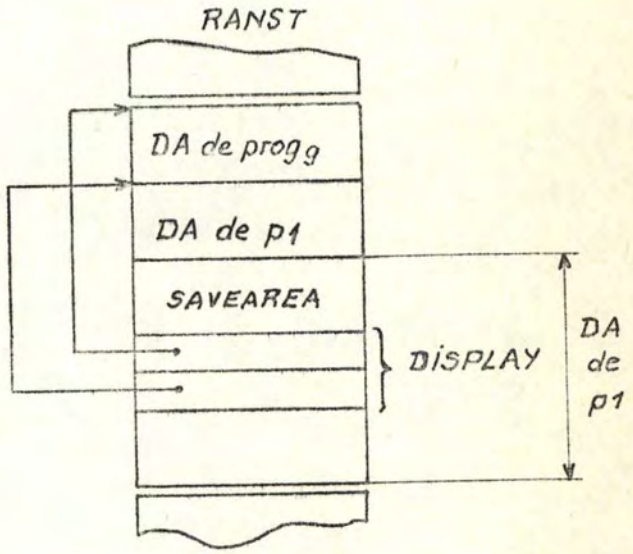
Cette zone "DISPLAY", est un ensemble de pointeurs. Chaque élément du "DISPLAY" pointe vers le début d'une "DA" qui se trouve plus bas sur la pile. Ces "DA" sont celles des procédures

dans lesquelles sont déclarées des variables utilisées dans notre procédure. C'est normal qu'on puisse retrouver la valeur des variables qu'on emploie dans une procédure. La profondeur de la procédure dans laquelle est déclarée une variable est connue statiquement, connaissant cela on peut statiquement savoir ou se trouve le pointeur d'une valeur à accès (n.p) (n sera la case n° x du "DISPLAY" ; x étant connu statiquement). La taille de "DISPLAY" est connue statiquement et est égale à quatre fois le nombre de pointeurs qui s'y trouvent.

Exemple :

Dans prog9 on a la procédure p2 qui emploie des variables qui sont déclarées en prog9 et en p1. Le "DISPLAY" compte deux éléments qui pointent vers les "DA" respectives. On le représente quand la "DA" de p2 est au sommet de RANST

```
prog9 : begin real y;
            proc p1 = void
            begin real x = 0;
                  proc p2 = void:
                  begin y+:=x ;
                  end ;
                  p2;
                  :
            end ;
      p1
end
```



(c) Deuxième zone de pointeurs : "DISDYN"

On a vu que les parties dynamiques des valeurs sont placées par bloc au sommet de DYNST. Le but de cette conception est de pouvoir récupérer aisément la place occupée par la partie dynamique des valeurs en sortant du bloc. La profondeur du bloc au sein de la procédure est une information statique, en sortant

d'un bloc on sait donc quel est la position relative de la zone de ses parties dynamiques sur DYNST. Il suffit de connaître l'adresse de début de chaque zone pour pouvoir réutiliser leur place en cas de suppression.

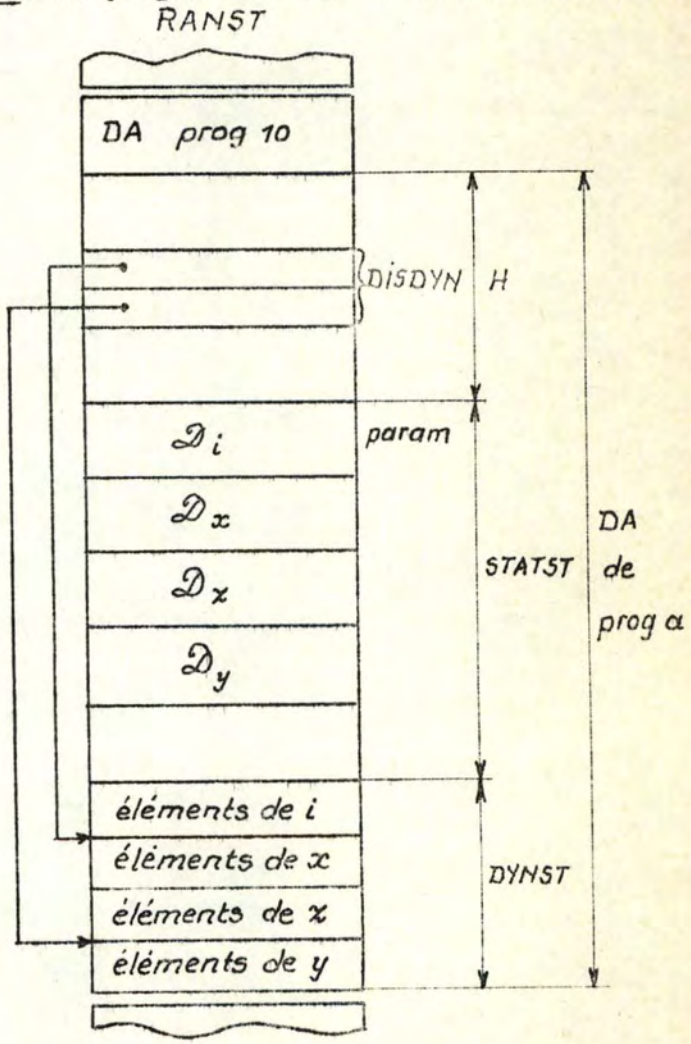
A cet effet on a introduit "DISDYN", c'est l'ensemble des pointeurs vers les sommets de la zone dynamique de chaque bloc. Le premier pointteur contient le début de la partie dynamique de bloc 1, le début de ce bloc étant le sommet de la partie dynamique des paramètres de la procédure (bloc zéro).

Les pointeurs de "DISDYN" doivent être mis à jour dynamiquement, leur valeur n'est connue qu'au moment où on connaît l'adresse du début de zone. "DISDYN" contient autant de pointeurs qu'il y a de blocs qui peuvent être en vie en même temps dans une procédure, soit ce nombre n, sa taille est alors de 4 x n.

Exemple :

On donne la configuration de la mémoire au moment ou on entre dans le bloc 2 de la proca de prog10. "DISDYN" a deux éléments dans ce cas ci.

```
prog10 :
 begin
     [ 1:2 ] int j ;
     proc a = ( [] int i) void :
        begin [ 1:3 ] real x, z ;
          begin [ 1:5 ] real y ;
          end
     end ;
   a (j)
 end
```

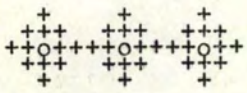


(d) Troisième zone de pointeur : "GC"

Dans cette zone on trouve toutes les informations dont on a besoin pour retrouver dynamiquement une variable qui est sur le "HEAP" au moment de l'appel du "garbage collector". On ne décrira pas cette zone, il suffit de savoir que sa taille est une information statique.

d. CONCLUSIONS

On peut conclure ce chapitre en disant que l'implémentation est orientée procédure. Les conventions de sauvetage d'information est faite conformément aux conventions IBM360 ce qui nous permet d'implémenter nos valeurs dans un environnement O/S de cette machine. En introduisant certaines informations supplémentaires dans l'en-tête (DISDYN) il nous est facile de gérer la récupération des parties dynamiques de valeurs en sortant d'un bloc.



CHAPITRE 3 : ETUDE APPROFONDIE DE L'ALGORITHME DE COPIE

a. ETUDE DES RESSOURCES

La description globale d'un compilateur a été faite dans l'introduction du mémoire. Nous savons que le compilateur traduit un programme écrit dans un langage source en un programme écrit en un langage objet qui est exécutable sur la machine considérée. La partie du compilateur qui nous intéresse dans ce mémoire, est la partie qui produit un programme objet qui peut copier une valeur d'un endroit de la mémoire vers un autre endroit pendant l'exécution du programme. Nous ne partons pas d'une instruction source pour faire la traduction, mais d'une instruction dans un langage intermédiaire. Dans notre cas cette instruction n'est autre qu'un appel de procédure du compilateur. Cette procédure est écrite en PL/I et a la syntaxe d'appel suivante : CALL CØPY (ADDRS, ADDRØ MØDE). Le compilateur complet est supposé être écrit en PL/I. PL/I ne dispose pas de zones "CØMMØN", nous les simulons en employant des variables externes qui existent pour la durée du "JOB". PL/I nous permet aussi de travailler avec des variables locales. Ce sera très utile lors d'appels récursifs de procédure.

Le compilateur complet n'existant pas encore, il nous faudra simuler certaines de ses parties pour pouvoir tester la génération de code symbolique.

La justesse du code même est testée à la main sur des exemples concrets. Les variables PL/I à simuler dans le programme de test sont : la table des modes, les variables globales de compilateur et les routines de travail.

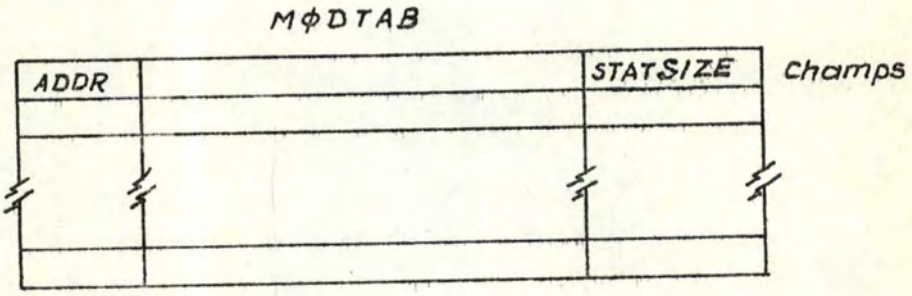
(1) *Table des modes*

A chaque instant du déroulement du programme qui génère le code pour la copie d'une valeur, il nous faut pouvoir connaître le mode. A cet effet on garde, pendant la compilation, une table des modes qui avait été crée bien avant que la procédure COPY en ait besoin. Nous simulons cette table par deux tableaux. Le premier tableau contient l'information statique de la valeur et le second décrit le mode.

(2) *Description des tableaux* (1)

Premier tableau : MØDTAB

A chaque entrée du tableau correspondent six champs. Nous en, décrivons le premier et le dernier qui nous intéressent dans notre problème. Ils sont respectivement un pointeur vers la table des descripteurs et la taille de la partie statique d'une valeur du mode décrit. Il y a autant d'entrées qu'il y a de valeurs de mode différent dans le programme.



ADDR : Valeur d'entrée dans le tableau DESTAB (voir ci-dessous)
Si la valeur est de mode de base il n'y a pas de pointeur.

STATSIZE : Taille de la partie statique de la valeur.

Deuxième tableau : DESTAB

A chaque entrée du tableau on peut avoir six descripteurs de mode différents. La forme du descripteur dépend du mode de la valeur décrite. La taille de chaque descripteur sera un nombre d'unités, chaque unité vaut une longueur du descripteur,

tous les champs sont supposés avoir la même taille. Les modes qui admettent des descripteurs sont : ref amode, [] amode, proc, struct, union et format (Ce dernier n'est pas décrit).

Le premier champ d'un descripteur est toujours, un code indiquant le mode de la valeur. Les codes utilisés dans ce travail sont :

---

(1) La description de la table des modes correspond à celle utilisée pour la future implémentation d'ALGOL68 à l'ERM (Ref de bibliographie 21)

MODE	CODE
ref	-1
proc	-2
format	-3
struct	-4
union	-5
rowof	-6

Les champs dans les quels on trouve amode sont des pointeurs vers la table MØDTAB.

Description des descripteurs

MODE	DESCRIPTEUR	TAILLE
ref amode	-1 | amode	2
[] amode	-6 | ND | amode NB : nombre de dimensions de la valeur multiple (≥ 1)	3
proc(bmode,... zmode) amode	-2 | amode | NP | bmode | ..... | zmode NP : nombre de paramètres de la procédure (≥ 0)	NP+3
union (amode,... zmode)	-5 | NA | amode | .... | zmode NA : nombre d'alternatives possibles de la valeur unitée (≥ 2)	NP+2

struct (amode a,... zmode z)

-4	NF	amode	a	....	zmode	z

2xNF+1

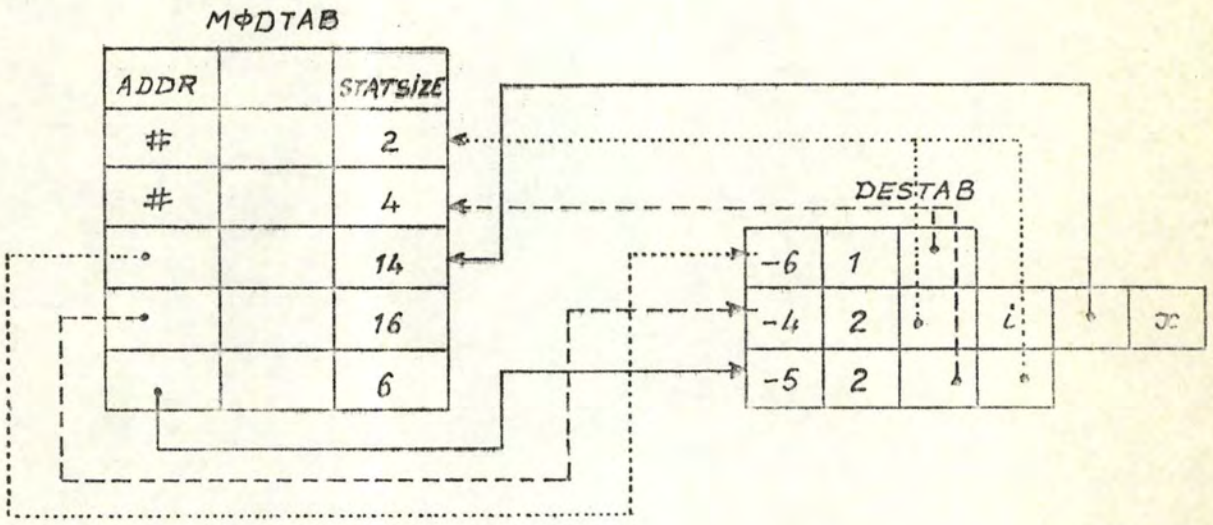
NF : nombre de champs de la structure

a,z pointeur vers la table de symboles (non décrite dans notre texte)

(b) Exemple :

Ayant un programme qui contient des valeurs de mode int, real, struct (int i,[]real x), union (real, int) on décrira la table MØDTAB et DESTAB correspondante.

Remarque : le signe "#" veut dire qu'il n'y a pas de pointeur



Nous gardons en PL/I nos tables sous forme de matrices.

MØDTAB : une entrée par mode

DESTAB : une entrée par champ.

MØDTAB

ADDR		STATSIZE
#		2
#		4
1		14
4		16
10		6

DESTAB

-6	1	2			
-4	2	1	i	3	x
-5	2	2	1		

Remarque : i et x ont une valeur numérique

(2) Variables globales du compilateur

LAB : génération d'étiquettes

DPRØC : profondeur statique de la procédure ou se trouve l'adresse objet de la valeur à copier.

NBLØC : profondeur du bloc dans cette procédure où se trouve cette adresse.

(3) Routines auxiliaires

Un compilateur est composé de plusieurs routines. Certaines nous intéressent pour notre travail. On en décrit systématiquement l'action, les paramètres et les particularités d'emploi. Le texte PL/I des routines se trouve en annexe (1a).

(a) Routine de génération de code : GEN

Action :

La routine génère du code sumbolique assembleur IBM360 à partir d'une matrice (des chaînes de caractères) qu'on passe comme paramètre.

Paramètres :

GENER : matrice ayant autant d'entrées qu'il y a d'instructions à générer

NUMB : nombre d'instructions à générer, le nombre s'écrit toujours en deux chiffres.

Particularités d'emploi

(i) Chaque chaîne de caractères contient trois champs (étiquette, code et opérandes). Les champs sont séparés l'un de l'autre par au moins un blanc, l'étiquette si elle existe commence au premier caractère, sinon elle doit être remplacée par au moins un blanc.

(ii) Quand on veut simuler une génération de code, on trouve systématiquement dans l'algorithme une pseudo-instruction PL/I. On donne en exemple ci-après le texte PL/I qui doit remplacer une pseudo-instruction correspondante. Ceci a été fait pour augmenter la lisibilité de l'algorithme en PL/I.

pseudo-instruction : GEN | texte en pseudo-assembleur | ;

pseudo-assembleur : une instruction en pseudo assembleur diffère d'une instruction réele par la possibilité d'apparition de variables du compilateur dans la partie étiquette ou opérande. Une variable compilateur y commencera toujours par le symbole "$".
donc LA 2,$N(2), ou $N = 2
donne comme code généré LA 2,2(2)

(iii) Exemple :

On décrit successivement la pseudo-instruction PL/I, le texte PL/I qui remplace la pseudo-instruction et le code génére.

pseudo instruction : GEN

```
L$LAB MVC  0($LEN,11),0(3)
      LA   11,$LEN(11)
      AH   3,12(2)
      BCT  8,L$LAB
```

Les variables compilateur $LAB et $LEN ont des valeurs respectives de 12 et de 8.

Texte PL/I :

```
DCL   TEXTE (4)  CHAR(40) ;
DCL   (LAB,LEN)  PIC'9999';
LAB = 12,  LEN = 8 ;
TEXTE (1) = 'L'||LAB||' MVC   0('||LEN||',11),0(3)';
TEXTE (2) = '           LA    11,'||LEN||'(11)';
TEXTE (3) = '           AH    3,12(2)';
TEXTE (4) = '           BCT   8',||LAB;
CALL GEN (TEXTE,04) ;
```

Remarque :

La dernière instruction est l'appel de la routine

GEN.

Code génére : L0012

```
MVC   0(0008,11),0(3)
LA    11,0008(11)
AH    3,12(2)
BCT   8,L0012
```

(b) Routine RELVNT

(i) : Appel CALL RELVNT (MØDEA, DYNREL, DYNSCØPE) ;

(ii) : Action : la routine va chercher dans la table des modes, pour un mode "MØDEA" bien défini, si une valeur de ce mode possède une partie dynamique, ou une portée qui nécessite une action de test de portée pendant la copie de la valeur.

(iii) : Paramètres

MØDEA mode, il est représenté ici par son entrée correspondante dans la table MØDTAB

DYNREL valeur binaire = '1'B s'il y a une partie dynamique
= '0'B s'il n'y a pas de partie dynamique

DYNSCØPE valeur binaire = '1'B s'il y a un test de portée dynamique
= '0'B s'il n'y a pas de test de portée dynamique.

Note : ce dernier nom est choisi ainsi parce que le test de portée qui est génére (si DYNSCOPE = '1'B) est un test dynamique.

(c) Routines de recherche dans la table des modes

Toutes les routines décrites ci-dessous, font des recherches dans la table des modes (MODTAB et DESTAB). Chaque routine reçoit en entrée le paramètre MØDE. Ce mode correspond au mode de la valeur sur laquelle on veut connaître des informations des tables. En PL/I le paramètre MØDE a une valeur qui correspond à l'entrée correspondante dans la table MØDTAB.

Les routines sont décrites tabulairement. Pour chaque

routine on trouve quatre explications : (1) le nom ; (2) les paramètres dans l'ordre obligatoire, (3) le code d'entrée ou de sortie (E ou S) , (4) la signification du paramètre. L'action faite par la routine n'est pas décrite, celle-ci découlant immédiatement de la signification des paramètres.

ROUTINE	*PARAM*	*E/S*	*SIGNIFICATION*
STSIZE	MØDE	E	-
	STATSIZE	S	Taille de la partie statique de la valeur considérée
MØDTYP	MØDE	E	-
	MØDETYP	S	Code correspondant à un des modes suivants: ref, format, proc, struct, union, [] amode
MODNXT	I	E	Numéro de séquence d'un champs d'une valeur structurée
	MØDE	E	-
	MØDEF	S	Mode du champ considéré
MDUNXT	I	E	Numéro correspondant à une alternative des valeurs possibles d'une valeur de mode union
	MØDE	E	-
	MØDEU	S	Mode de l'alternative considérée
MODERW	MØDE	E	-
	MØDER	S	Mode d'un élément d'une valeur multiple
MODNMB	MØDE	E	-
	NUMB	S	Nombre de champs d'une valeur structurée ou nombre d'alternatives possibles d'une valeur de mode union.

(d) Routine ADRIBM

(i) : Appel : CALL ADRIPM (REG, ADD) ;

(ii) : Jusqu'ici on n'a pas encore utilisé l'adresse machine. La routine ADRIBM génère du code tel que l'adresse représentée dans notre programme par le couple "n,p" soit traduit pendant l'exécution de la copie en adresse machine. Cette adresse machine sera placée dans le registre générale REG.

(iii) : Paramètres :

ADD : adresse dans le compilateur, étant le couple n,p. Déclaration PL/I de ADD : DCL 01 ADD, 02 (N,P) BIN FIXED ;

REG : numéro du registre général ($0 \leq REG \leq 15$)

(e) Routine CØPYST

(i) : Appel : CALL CØPYST (RS, RØ, SIZE) ;

(ii) : la routine génère du code pour copier un nombre (SIZE) de caractères qui sont consécutifs en mémoire. L'adresse source et objet de cette copie se trouvent respectivement dans les registres généraux RS et RØ.

(iii) : Paramètres

RS : $0 \leq RS \leq 15$
RØ : $0 \leq RO \leq 15$
} $RS \neq RØ$

SIZE : Nombre de caractères à copier. $1 \leq size \leq 32767$

(iv) : Particularités :

Le code généré par cette routine doit être très efficient. Pour cette raison, les instructions assembleur générées dépendent de la valeur de "SIZE" au moment de l'appel de CØPYST. Si SIZE ≤ 256, ou génère une instruction assembleur symbolique :

" MVC O(RØ, SIZE),O(RS)".

Cette instruction a comme action de copier un nombre de size caractères consécutifs en mémoire d'une adresse source vers une adresse objet. Ces adresses se trouvent respectivement dans les régistres généraux RS et RØ.

Si SIZE > 256, on génère une boucle qui copie le bloc de caractères par 256.

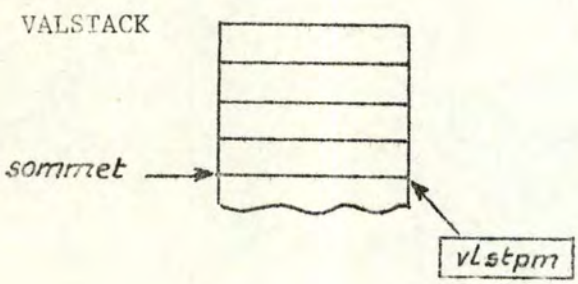
b. ETUDE DE LA GENERATION DE CODE CORRESPONDANT A UN TERMINAL DE LA GRAMMAIRE GENERATIVE (1.f.(3) p14.

(1) *Introduction*

Avant de décrire la génération de code proprement dite, on s'arrête à la notion de récursivité. La notion mode est récursive. L'algorithme de copie que nous avons décrit l'est aussi. Cette récursivité ne donne pas de problèmes dans l'algorithme de génération, car le langage PL/I admet la récursivité. Le code généré par contre, assembleur symbolique, n'admet pas la récursivité. Les actions qui se font pendant la copie d'une valeur sont exécutées dans le même ordre qu'apparait le code correspondant à la chaîne terminale de sortie.

Si on regarde l'arbre grammatical correspondant à une valeur à copier, on voit immédiatement qu'il est parfois nécessaire de repasser par un noeud qui se trouve plus près du sommet pour pouvoir atteindre un autre terminal. En général on ne peut pas remarquer ce va et vient dans l'arbre grammatical au niveau de l'exécution de la copie. Cependant, au moment ou on copie une valeur qui a une partie dynamique on se met dans une condition fixe ; on réserve deux registres généraux qui pointent respectivement vers la partie statique et vers la partie dynamique. Au moment ou l'on commence la copie de la partie dynamique des éléments de la valeur on doit se remettre dans les mêmes conditions (qu'avant) cela veut dire que pendant cette copie les mêmes régistres pointent vers la partie statique et dynamique de l'élément.

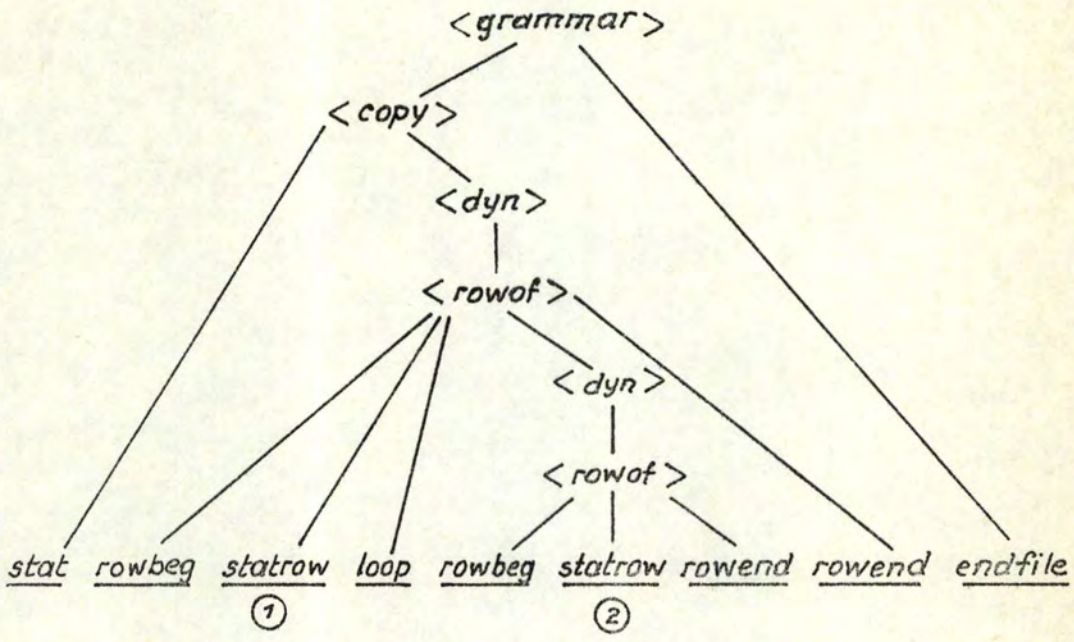
Pour faire tous ces sauvetages on a crée une pile VALSTACK pendant l'exécution, on y place au sommet les valeurs qui sont a sauver pendant qu'on descend dans l'arbre et on les reprend du sommet quand on remonte l'arbre.



- Chaque élément est de taille 4
- le sommet est pointé par vlstpm
- la taille maximale de VALSTACK peut être calculée pendant la compilation.

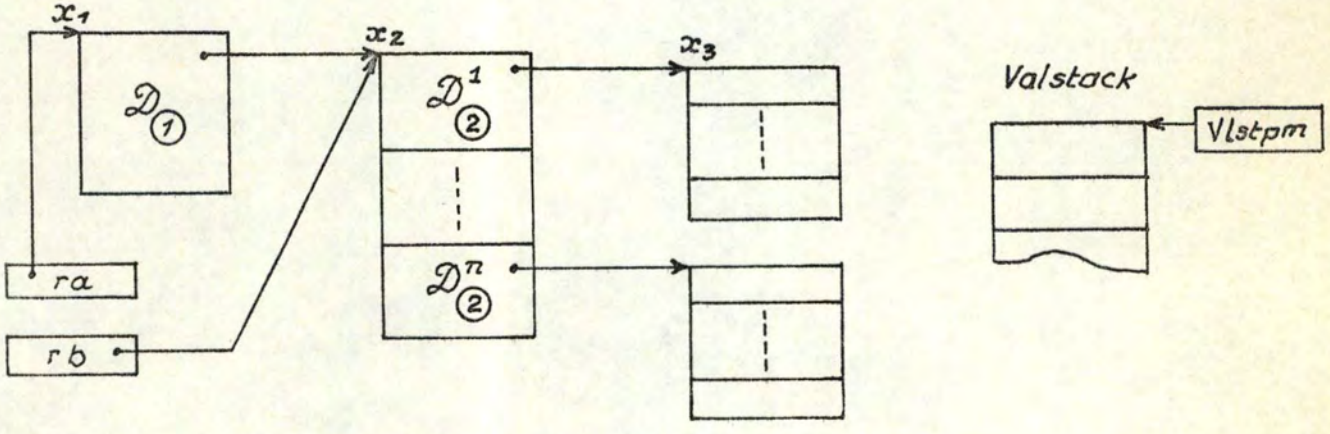
Exemple :

Copie de [1:n] [1:n] real

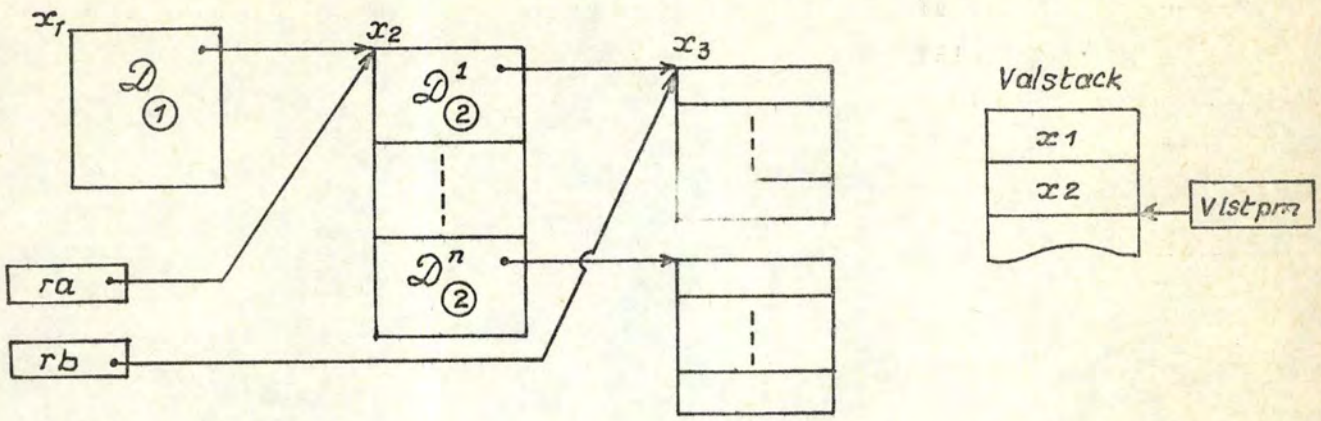


Les deux instants qui nous interessent pendant la copie pour expliquer notre sauvetage de régistres sont ceux ou on copie la partie dynamique des éléments d'une valeur multiple. Cela se retrouve toujours dans la chaîne de sortie par le terminal statrow. Il y a deux terminaux statrow, puisqu'il y a deux profondeurs de valeur multiple. La figure ci-dessous montre la configuration des pointeurs au moment 1 et 2.

① Copie des descripteurs des éléments ② (partie dynamique de ①)



2. Copie des éléments real (partie dynamique de ②)



(2) *Routines de sauvetage*

SAVREG (N, REG) : génère du code pour sauver le contenu du registre général REG au sommet de VALSTACK.

RESREG (N, REG) : génère du code pour restituer le contenu du registre général REG à partir du sommet de VALSTACK

Paramètres :

N : nombre entier dont la valeur détermine la génération du code au non. Dans certains cas (ou la verra plus bas) on ne doit pas sauver de registres.

REG : nombre entier : 0 ≤ REG ≤ 15

(3) *Etude du terminal stat*

L'apparition du terminal stat dans la chaîne terminale de production ; correspond à la génération de code qui effectue la copie de la partie statique de la valeur. La taille de cette partie statique est connue statiquement. On peut donc générer un code optimal pour effectuer cette copie.

On peut remplacer stat par trois appels de routines

```
CALL  ADRIBM  (RS, ADDS)
CALL  ADRIBM  (RØ, ADDØ)
CALL  CØPYST  (RS, RØ, SIZE)
```

(4) *Etude du terminal scope*

Nous avons fait une étude de portée en (1 C (1) p 6) et nous en

avons décrit l'implémentation en (2 B (2) p24).

Rappelons que la portée est gardée en mémoire sous forme d'un couple (np, nb). Au moment où nous décidons d'effectuer une copie de valeur, nous connaissons l'emplacement précis de cette copie. Nous en connaissons donc le (n'p, n'b).

n'p : profondeur de la procédure ou se trouve l'adresse objet

n'b : profondeur du bloc dans la procédure ou se trouve cette adresse.

On peut remplacer scope par des instructions qui comparent le couple (np,nb) avec (n'p, n'b), il faut toujours (np, nb) soit plus grand ou égal à (n'p, n'b) (on compare la concaténation du couple (np, nb) avec celle du couple (n'p, n'b)), si ce n'est pas le cas une routine d'erreur est appelée. On peut simuler cette génération de code par la pseudo-instruction PL/I suivant :

GEN

```
IF (NP N'P) | ((NP=N'P) & (NB<N'B))
    THEN CALL Routine d'erreur
```

(5) Etude du terminal statrow

Le terminal statrow correspond toujours à la génération de code qui copie la partie statique des éléments d'une valeur multiple. Ces éléments peuvent être consécutifs en mémoire ou non. Le procédé de copie est simple quand les éléments sont consécutifs, il l'est moins dans le cas contraire. Malheureusement on ne peut décider de la consécutivité ou non pendant la compilation. On ne peut le savoir que pendant l'exécution (sauf optimisation dans des cas simples).

Pour chaque copie d'une valeur multiple, on génère du code pour les deux cas. Le code pour copier des éléments non consécutifs suffiront, mais pour des raisons d'optimisation pendant l'exécution on y ajoute le code pour copier les éléments consécutifs. Le code qui est exécuté est choisi en fonction de la valeur flag dans le descripteur (2b (1) (d) p27)

(a) Les éléments sont consécutifs en mémoire

On ne peut pas générer le code de copie de façon définitive,

car on ne sait pas combien d'élément possède la valeur multiple. Ce nombre d'éléments est connu pendant l'exécution. Dans ce cas nous n'avons besoin de connaître pendant l'exécution que le nombre total de caractères à déplacer (encombrement des parties statiques de la partie dynamique). On trouve cet encombrement dans le descripteur, c'est le champ "d_o". La stratégie de copie est la suivante : on copie par blocs de 256 caractères jusqu'au moment ou on a copié tout. Le dernier bloc copié peut être moins long. Descripteur de l'algorithme de copie (description faite en pseudo PL/I).

ADDS : adresse source

ADDØ : adresse objet

```
L : IF TAILLE = 256
      THEN Copier "taille" caractères de ADDS vers ADDØ ;
      ELSE DØ copier "256" caractères de ADDS vers ADDØ ;
             TAILLE = TAILLE - 256 ;
             ADDS   = ADDS   + 256 ;
             ADDØ   = ADDØ   + 256 ;
             GOTØ L ;
           END
```

Nous avons crée une routine de travail qui génère le code correspondant à l'algorithme décrit ci-dessus. C'est la routine COPYD2.

COPYD2 (RS, RØ)

Appel : CALL CØPYD2 (RS, RØ)

Paramètres :

RS : régistre de base dans lequel on trouve l'adresse source machine

RØ : régistre de base dans lequel on trouve l'adresse objet machine

Remarque :

La routine génère aussi le code pour prendre la taille à copier "d_o", dans le descripteur.

(b) Les éléments ne sont pas consécutifs en mémoire

Quand les éléments d'une valeur multiple ne sont pas consécutifs en mémoire (la valeur est une sous-valeur multiple).

Il nous est impossible de les copier tous à la fois. On doit copier élément par élément. Il nous faut donc pouvoir parcourir tous ces éléments. La stratégie de parcours choisie doit être efficace, car cette stratégie résulte en code qui est exécuté pendant la copie de la valeur. Nous étudions successivement trois stratégies qu'on compare entre-elles pour choisir la stratégie définitive.

Chaque stratégie se compose de deux parties. Une initialisation et une continuation. La partie initialisation positionne un pointeur sur le premier élément à copier, et initialise certaines constantes de travail indispensables au parcours. La partie continuation fait la recherche de l'élément suivant à copier, et décide de la fin de parcours. L'exécution totale d'une stratégie est toujours la même :

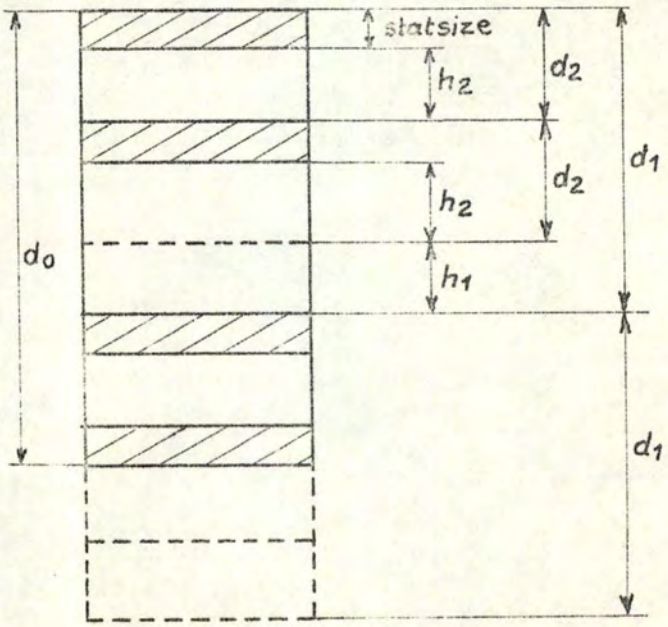
1. initialisation
2. copier un élément
3. continuation
   - Si dernier élément - continuer
   - Sinon aller en 2

1° Stratégie

- Configuration mémoire des éléments

On représente les éléments d'une valeur de mode [1:2, 1:2] amode.

Chaque élément de taille "STATSIZE" est représentée par une partie hachurée de la figure.



d_i : taille totale d'un élément de dimension i (2b (2) (d) p28)

h_i : taille d'un trou laissé entre deux éléments de dimension i

d_o : encombrement total des parties statiques des éléments.

On crée un compteur courant dans chaque dimension : x_i.

Ce compteur est initialisé à la borne inférieure de chaque dimension.

A chaque instant les x_i ont comme valeur les indices de l'élément qu'on va copier. Si on copie l'élément a[1,1] alors $x_1 = 1$ et $x_2 = 1$; pour l'élément a[2,1] $x_1 = 2$ et $x_2 = 1$.

En général pour copier l'élément a[i,j,...,1) ; $x_1 = i$, $x_2 = j$ et $x_n = 1$.

$x_i : \Rightarrow l_i \leq x_i \leq u_i \quad ; i \in [1,n]$

Algorithme

Dans la partie initialisation toutes les valeurs x_i et h_i sont calculées à partir des valeurs l_i, u_i et d_i du descripteur de la valeur multiple. On calcule l'adresse du premier élément (add).

Dans la partie continuation ou incrémente le compteur x_i de la dernière dimension, si celui ci est égal à u_i on incrémente le x_i de la dimension précédente et on remet le dernier à l_i. Si tous les x_i sont égaux aux u_i on copie la dernière valeur et on ne fait rien. On remet aussi à jour le pointeur vers l'élément suivant.

Initialisation

```
X(N) = L(N);
H(N) = D(N) - STATSIZE ;
DØ I = N-1 TØ 1 BY - 1 ;
   X(I) = L(I);
   H(I) = D(I) - (U(I+1) - L(I+1)+1) * D(I+1);
END ;
ADD = POINTEUR ;
L : copie de l'élément
```

x(I)	x_i
L(I)	L_i
U(I)	U_i
D(I)	d_i
POINTEUR	pointeur dans le descripteur.

Continuation

```
DØ  I = N TØ 1  BY - 1;
  IF X (I) = U(I);
     THEN X(I) = L(I);
     ELSE DØ; X(I) = X(I) + 1;
              INTER = STATSIZE;
              DØ J = I TØ N ;
                  INTER = INTER + H(J);
              END;
              ADD = ADD + INTER;
              GØ TØ L;
          END ;
END;
```

Calcul du nombre d'opérations effectuées

soit : $d_i = u_i - l_i + 1$

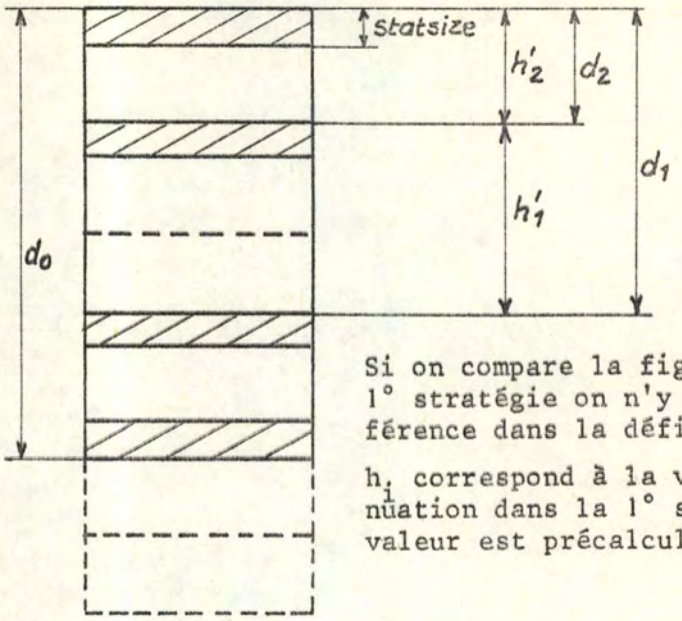
n = nombre de dimensions de la valeur

On donne pour chaque partie le nombre d'affection (=), de sommes (+,-) et de produits (*) et (test).

	Initialisation	Continuation
(=)	$3n$	$2\sum_{i=1}^{n-1}\left(\prod_{j=1}^{i} d_j\right) + 4\prod_{i=1}^{n} d_j + 2\sum_{k=0}^{n-1}(n-k)\left\{\prod_{j=1}^{k+1} d_j - \prod_{j=1}^{k} d_j\right\} + 2$
(+,-)	$7n - 6$	$\sum_{i=1}^{n}\left(\prod_{j=1}^{i} d_j\right) + 2\left(\prod_{j=1}^{n} d_j - 1\right) + 2\sum_{k=0}^{n-1}(n-k)\left\{\prod_{j=1}^{k+1} d_j - \prod_{j=1}^{k} d_j\right\}$
(*)	$n - 1$	–
(test)	–	$\sum_{i=1}^{n}\left(\prod_{j=1}^{i} d_j\right)$

2° Stratégie

Configuration mémoire des éléments



Si on compare la figure à celle de la 1° stratégie on n'y voit qu'une différence dans la définition de h_i.

h_i correspond à la valeur INTER (continuation dans la 1° stratégie). Cette valeur est précalculée à présent.

- Algorithme

Voir celui de la première stratégie, sauf qu'on précalcule en plus les valeurs h'_i.

Initialisation HP(I) : h'i

```
X(N) = L(N) ;
HP(N)= D(N) ;
DØ I = N-1  TØ 1 BY - 1;
  X(I) = L(I);
  HP(I) = HP(I+1) + D(I) - (U(I+1) - L(I+1) * D(I+1);
END;
ADD = PØINTEUR ;
L : copie de l'élément
```

Continuation

```
DØ I = N TØ  1  BY - 1 ;
  IF X(I) = U(I)
     THEN X(I) = L(I) ;
     ELSE DØ; X(I) = X(I) + 1 ;
              ADD = ADD + HP(I);
              GØTØ L;
          END;
END;
```

Calcul du nombre d'opérations $u_i - l_i + 1$

	Initialisation	Continuation
(=)	$3n$	$2\sum_{i=1}^{n-2}\left(\prod_{j=1}^{i} d_j\right) + 3\prod_{i=1}^{n} d_i - 1$
(+,-)	$g(n-1)$	$\sum_{i=1}^{n}\left(\prod_{j=1}^{i} d_j\right) + 2\left(\prod_{i=1}^{n} d_i - 1\right)$
(*)	$n-1$	–
(test)	–	$\sum_{i=1}^{n}\left(\prod_{j=1}^{i} d_j\right)$

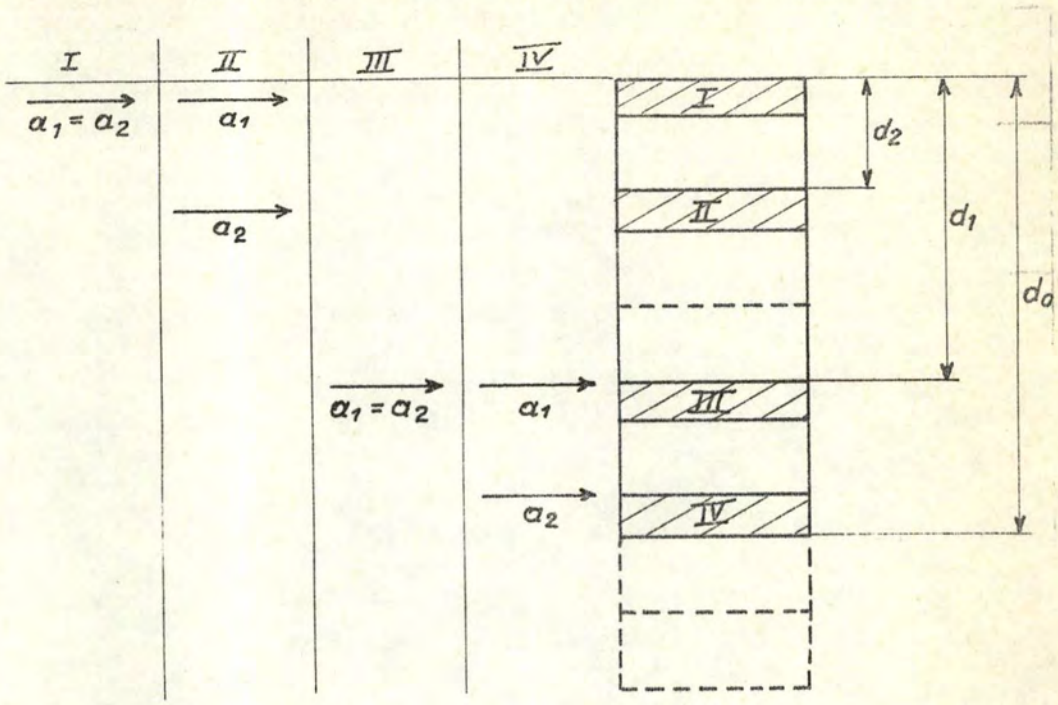
3° Stratégie

Cette stratégie n'utilise pas la taille des trous pour mettre à jour le pointeur courrant après une copie. On utilise un pointeur courant par dimension pour simuler ce mécanisme. La distance entre le début de deux sous-valeurs de dimension i est "di" (la taille de cette sous-valeur). On ne fait que garder chaque pointeur au début de la sous-valeur dans laquelle on copie un élément. Chaque fois qu'on a copié tous les éléments d'une sous-valeur de dimension i, il faut remettre à jour le pointeur correspondant, il pointera vers la sous-valeur suivante. Les pointeurs des sous-valeurs intérieures à la sous-valeur en question seront égalisées au pointeur de la dimension i.

Dans la figure suivante on montre la valeur des pointeurs pour la copie de chacun des éléments.

Configuration mémoire des éléments

Place des pointeurs pendant la copie des éléments



a_i = pointeur de la dimension i

Algorithme

Dans la partie initialisation, on initialise les valeurs x_i (voir 1° stratégie) et les pointeurs a_i

Dans la partie continuation on incrémente ces pointeurs, si tous les éléments sont copiés on termine, sinon on copie et on recommence la continuation.

Initialisation

```
DØ  I = 1 TØ  N-1 ;
   A(I) = PØINTEUR;
   X(I) = L(I) ;
END;
ADD = PØINTEUR;
X(N) = L(N);
```

POINTEUR	pointeur du descripteur
A(I)	a_i

L : copie de la valeur

Continuation

```
IF   X(N) ¬= U(N)
   THEN DØ ; ADD = ADD + D(N);
             X(N)=X(N) + 1;
             GØTØ L;
        END;

DØ I = N-1 TØ  1  BY - 1;
  IF X(I) = U(I)
     THEN X(I) = L(I) ;
     ELSE DØ ; X(I) = X(I) + 1;
               A(I) = A(I) + D(I);
               DØ J = N-1  TØ  I+1  BY - 1;
                  A(J) = A(I);
               END;
               ADD = A(I);
               X(N) = L(N);
               GØTØ L;
          END;
END ;
```

Calcul du nombre d'opération ($d_i = u_i - 1_i + 1$)

	Initialisation	*Continuation*
(=)	$3n-1$	$2n+2\left(\prod_{i=1}^{n} d_i - \prod_{i=1}^{n-1} d_i\right)+7\sum_{i=1}^{n}\left(\prod_{j=1}^{i} d_j - 1\right)+2\sum_{i=1}^{n-2}\left[i * \prod_{j=n-i}^{1}(d_j-1)\right]$
(+)	$n-1$	$n+2\left(\prod_{i=1}^{n} d_i - \prod_{i=1}^{n-1} d_i\right)+4\sum_{i=1}^{n}\left(\prod_{j=1}^{i} d_j - 1\right)+\sum_{i=1}^{n-2}\left[i \prod_{j=n-i}^{1}(d_j-1)\right]$
(*)	–	–
(test)	–	$2\sum\left(\prod_{j=1}^{n} d_j\right)+\prod_{i=1}^{n} d_i$

Comparaison des stratégies

Pour comparer les deux stratégies, on ne compare pas les initialisations (nombre restreint d'opération et non repris dans une boucle). On compare donc les parties continuation.

Nous ne comparons pas la 1° stratégie avec la 2° stratégie, car la deuxième étant une simplification de la première donne toujours moins d'opérations. On fait donc la comparaison entre la 2° et la 3° stratégie pour les opérations (=); (+,-). (Le nombre de (*) sont les mêmes pour les deux stratégies, et le nombre de (test) est plus grand pour la troisième). Pour avoir une idée de l'évalution du nombre d'opérations à exécuter pendant la copie, on compare les deux stratégies pour des valeurs de n (nombre de dimensions) n = 1,2 et pour des valeurs d_1, d_2, de pouvant être 2 ou 5.

Exemple : n= 2 ; d_1 = 5 ; d_2 = 5 =>[4:8,13 :17] real x;

Tableau de comparaison

n	d_1	d_2	*2° stratégie*		*3° stratégie*	
			(=)	*(+,-)*	*(=)*	*(+,-)*
1	2	-	5	4	13	9
	5	-	14	13	40	27
2	2	2	15	12	40	22
	2	5	35	30	90	58
	5	2	39	33	105	64
	5	5	84	78	240	154

On voit immédiatement que la 3° stratégie a tendance a demander plus d'opérations. Pour cette raison nous avons choisis la 2° stratégie pour parcourir une valeur multiple. Le code qui est généré pour l'initialisation et la continuation, l'est fait par l'intermédiaire de deux routines PL/I qui sont respectivement : INITAL et CONTIN. Ces deux routines n'ont qu'un paramètre : la dimension de la valeur multiple.

(5) Etude du termonal loop

Le terminal loop correspond à la génération de code qui copie la partie dynamique des éléments d'une valeur multiple. Il faut que pour chaque élément on puisse se mettre dans les conditions de départ, c'est à dire qu'un régistre pointe vers la partie statique de l'élément et qu'un autre pointe vers la partie dynamique. On fera donc un sauvetage des régistres pour chaque copie. Pour pouvoir copier les parties dynamiques, il faut générer une boucle.

On génère ce qui suit :

L : Sauver les registre Ra et Rb
Mettre à jour Ra et Rb pour la copie de l'élement suivant — A

Copier la partie dynamique — B

Restituer les registre
Tester si toutes les parties dynamiques ont été copiées — C
si non => L
si oui =>fini

Le code de A et de C se trouve en annexe.

B se traduit par l'appel d'une routine CPDNPT qu'on explique plus tard.

c. ETUDE DE L'ALGORITHME PL/I DE GENERATION DE CODE

L'étude de l'algorithme de génération de code, correspond essentiellement à l'étude des non-terminaux de la grammaire générative. Si la chaîne terminale de cette grammaire correspond aux actions qui sont à prendre pendant la copie d'une valeur, l'arbre syntaxique nous montre ce que fait le compilateur pour pouvoir générer ce code.

Nous ne décrirons pas le non-terminal < grammar >. Il est introduit dans la grammaire pour pouvoir le tester.

Les autres non-terminaux seront décrits successivement. A chaque non-terminal correspond une procédure PL/I. Les paramètres des procédures sont une ou deux adresse (n,p) et un mode (entrée dans la table MØDTAB).

(1) <Copy >

Ce non-terminal a l'appel CØPY (ADDS, ADDØ, MØDE);

La procédure CØPY génère le code pour la copie de la partie statique de la valeur (terminal stat) et appelle ensuite une autre procédure qui traite la partie dynamique de la valeur.

TEXTE PL/I

```
CØPY : PRØC (AD1, AD2, MØDE);
  CALL STATSIZE (MØDE, STATSIZE) : /* recherche de la taille statique
                                      de la valeur                    */
  CALL ADRIBM (R2,AD1);            /* on génére le code qui place AD1 */
  CALL ADRIBM (R3,AD2);            /* et AD2 dans R2 et R3            */
  CALL COPYST (R2, R3,STATSIZE) ;  /* générer le code pour la copie   */
                                   /* de la partie statique           */
  IF STATSIZE <= 4  THEN RETURN;   /* si la taille statique≤ 4 on ne  */
                                   /* peut pas avoir une partie dy-   */
                                   /* namique, c'est fini.            */
  CALL RELVNT (MØDE, DYNREL,       /* y-a-t-il une partie dynamique   */
                     DYNSCOPE);    /* ou une portée.                  */
  IF DYNREL = '0'B & DYNSCOPE = '0'B /* s'il n'y a ni partie dynamique */
  THEN RETURN;                     /* ni portée, c'est fini           */

  /* Instructions qui calculent la profondeur de la */
  /* procédure et du bloc au sommet de la pile      */

  CALL CPDNPT (AD2,MØDE) ;         /* il y a une partie dynamique ou  */
                                   /* une portée, on appelle une autre*/
                                   /* procédure.                      */
END COPY ;
```

(2) <Dyn >

Comme on a pu voir dans la grammaire, la procédure CPDNPT (ADD, MØDE) qui correspond à ce terminal, ne sert qu'à faire, un choix d'une procédure qu'on va appeler, ou bien, de générer du code pour faire un test de portée.

Texte PL/I

```
CPDNPT : PRØC (ADD, MØDE);
  CALL MØDTYP (MØDE,MØDEX);        /* MØDEX est un code correspondant */
                                   /* à un mode :                     */
                                   /*      ref     : 1                */
                                   /*      proc    : 2                */
```

```
                                        /*    format    : 3              */
                                        /*    struct    : 4              */
                                        /*    union     : 5              */
                                        /* [] amode     : 6              */
  IF MØDEX >= 1 & MØDEX <= 3
    THEN DØ;  GEN
                        code pour effectuer le test dynamique de
                        portée                                        ;
              RETURN;
         END;
  CALL SAVREG (N,R2);                   /* générer le code pour sauver le */
                                        /* régistre R2                    */
  CALL ADRIBM (R2,ADD);                 /* générer le code pour la mise à */
                                        /* jour de R2                     */
  IF MØDEX = 4 THEN CALL STRUCT (ADD,MØDE);
  IF MØDEX = 5 THEN CALL UNIØN  (ADD,MØDE);
  IF MØDEX = 6 THEN CALL RØWØF  (ADD,MØDE);
  CALL RESREG (N,R2) ;                  /* générer le code pour restituer */
                                        /* R2                             */
END CPDNPT;
```

(3) <*Struct*>

Ce non-terminal correspond à la procédure STRUCT (ADD,MØDE) qui est appelé par CPDNPTsi la valeur à copier est de mode struct et a une partie dynamique ou une portée.

TEXTE PL/I

```
STRUCT : PRØC (ADD,MØDE);
 CALL MODNMB (MØDE, NUMFIELDS) ; /* recherche du nombre de champs de */
 /* ADD1 = ADD */                /* la structure                     */
 DØ I = 1 TØ NUMFIELDS ;
    CALL MØDNXT (I, MØDE, MØDEF);/* Quel est le mode du champ  n° I  */
    CALL RELVNT (MØDE, DYNREL,   /* Ce champ  a-t-il une partie dy-  */
                 DYNSCOPE);      /* namique ou une portée            */
    CALL STSIZE (MØDEF, STATSIZE )/* taille de la partie statique du */
                                 /* champ.                           */
    IF DYNREL = '1'B | DYNSCOPE = '1'B
       THEN CALL CPDNPT (ADD1, MØDEF) ;
    /* ADD1 = ADD1 + STATSIZE */
 END ;
 END STRUCT;
```

(4) <*union*>

Ce non-terminal correspond à la procédure UNIØN (ADD,MØDE), qui est appelée par CPDNPT si la valeur à copier est de mode union et a une partie dynamique ou une portée.

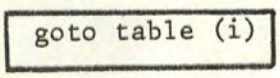
Le code à exécuter n'est connu qu'au moment de l'exécution. Il dépend du mode à l'exécution. Dans l'en-tête de la valeur, on trouve un code correspondant à ce mode. Pendant la génération de code, nous devons prévoir du code pour la copie de toutes les alternatives de la valeur (si une alternative n'a ni partie dynamique, ni portée, on ne génère rien), et c'est à l'exécution qu'on branchera vers le bon code. Pour pouvoir effectuer ce branchement, on tient une table de branchements. Il y a autant d'entrées que d'alternatives, et à chaque entrée dans la table on trouve une instruction de branchement vers le code correspondant. Au début de la copie on doit se brancher, en fonction de la valeur qui se trouve dans l'en-tête, vers la bonne entrée dans la table.

Si à l'exécution de la copie l'alternative i correspond au mode actuel on fait un saut vers l'entrée i de la table. A partir de cette entrée on saute vers le code de copie correspondant.

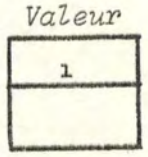
Exemple :

Soit à générer du code pour copier une valeur de mode union (int,[]real, struct ([]int, real), real,[] char) ;

La valeur a 5 alternatives. Soit l'alternative i celle en cours au moment de la copie.

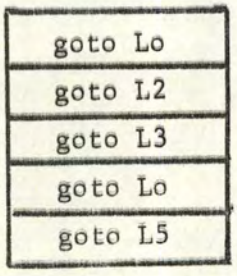


Valeur



en-tête

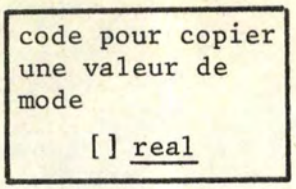
table



Remarque :

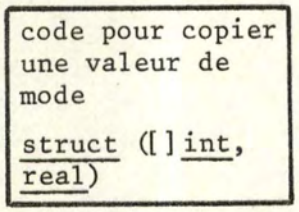
Pour les alternatives 1 et 4 il n'y a pas de code à exécuter.

L2 :



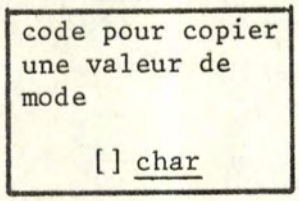
goto Lo

L3 :



goto Lo

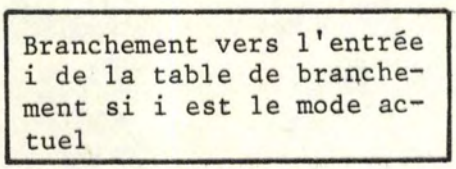
L5 :



Lo :

TEXTE PL/I

```
UNIØN : PRØC (ADD,MØDE);
    CALL MØDNMB (MØDE, NUMMØDE);     /* recherche du nombre d'alterna- */
                                      /* tives de la valeur             */

    GEN
```



;

```
    DØ I = 1 TØ  NUMMØDE ;
          CALL MDUNXT (I, MØDE, MØDEU) ; /* mode de l'alternative I */
          CALL RELVNT (MØDEU, DYNREL, DYNSCØPE);
          IF DYNREL = '1'B | DYNSCØPE = '1'B
             THEN GEN [GØTØ LI] ;
             ELSE GEN [GØTØ LO] ;

    END ;
```

```
DØ  I = 1 TØ NUMMØDE;
    CALL MDUNXT (I,MØDE,MØDEU);
    CALL RELVNT (MØDEU, DYNREL, DYNSCØPE);
    IF DYNREL = 1'B| DYNSCØPE = '1'B
      THEN DØ ; [GEN   LI :]   ;
                CALL CPDNPT (ADD1, MØDEU); /* ADD1 est     */
                                           /* une adres-   */
                                           /* se du début  */
                                           /* de la va-    */
                                           /* leur l'en-   */
                                           /* tête         */

                GEN [GØTØ LO] ;
           END ;
    END ;
    GEN [LO :] ;
END UNIØN;
```

(5) <*rowof*>

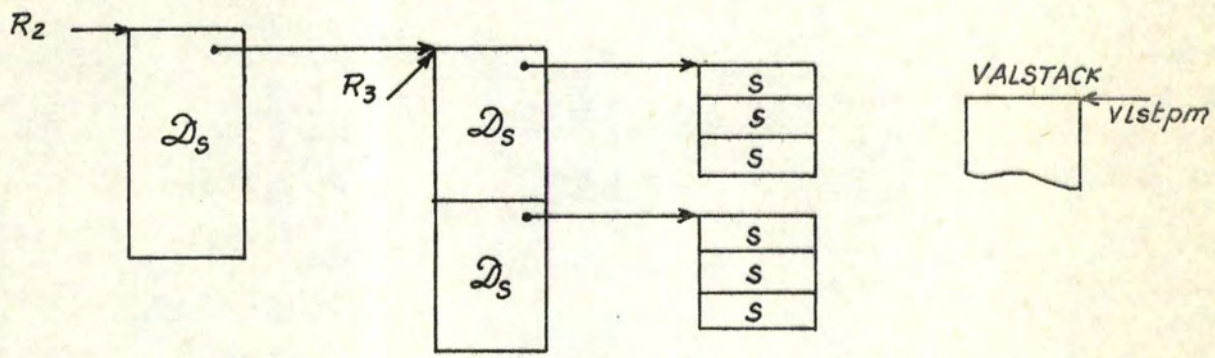
Ce non-terminal correspond à la procédure RØWOF(ADD,MØDE); qui est appelée par CPDNPT si la valeur à copier est une valeur multiple. A part le module CØPY, RØWOF est le seul module à générer du code de copie. On y retrouve la génération de code qui copie les parties statiques des éléments. Le code est généré pour le cas ou les éléments sont consécutifs en mémoire ou non. Ensuite on y retrouve le code pour faire la copie successive de la partie dynamique de chaque élément.

Avant de générer du code pour la copie des parties statiques des éléments. Il faut en générer pour calculer la place réquise sur la pile. S'il ne reste plus assez de place on appelle le GARBAGE-CØLLECTØR. Si les éléments ne sont pas consécutifs, il faut calculer une nouvelle valeur d_o. Les éléments sont compactés durant la copie. Après la copie il faut mettre à jour le descripteur si les éléments ont été compactés.

Nous allons d'abord donner schématiquement toutes les étapes d'une copie d'une valeur de mode [1:2][1:3] amode. Le VALSTACK y sera représenté avec son pointeur du sommet vlstpm. Le pointeur de

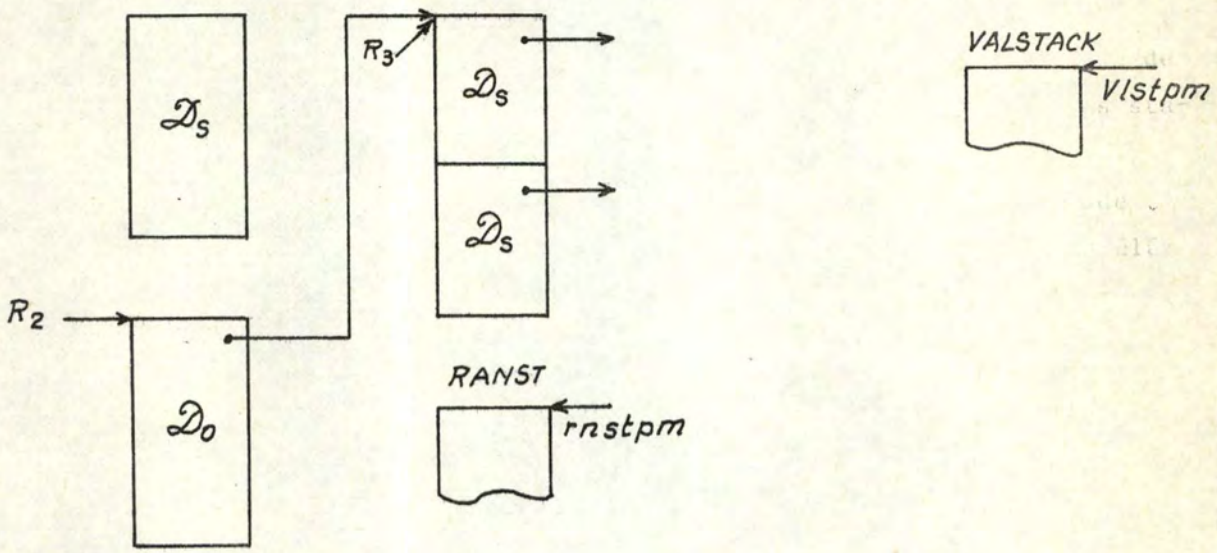
sommet de la pile est rnstpm. Le régistre de base R2 pointe toujours vers la partie statique traitée et le régistre de base R3 pointe toujours vers la partie dynamique traitée.

① Valeur avant la copie



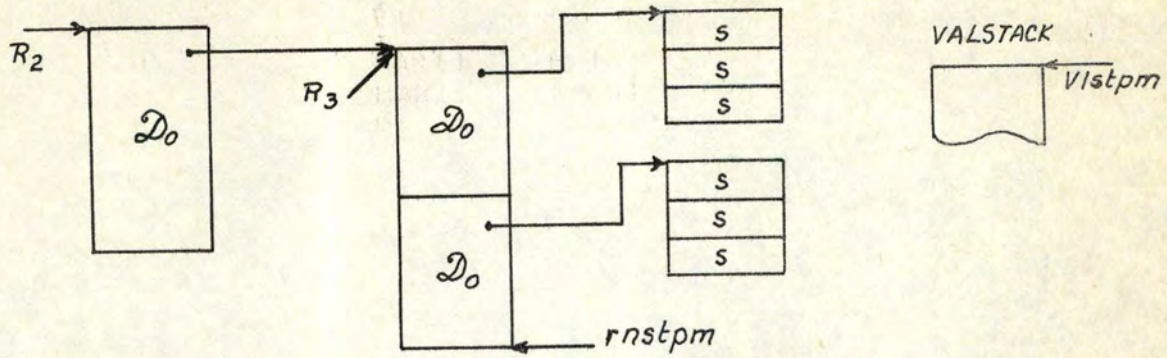
D_s : Descripteur source
s : Elément source

② Valeur après la copie de la partie statique

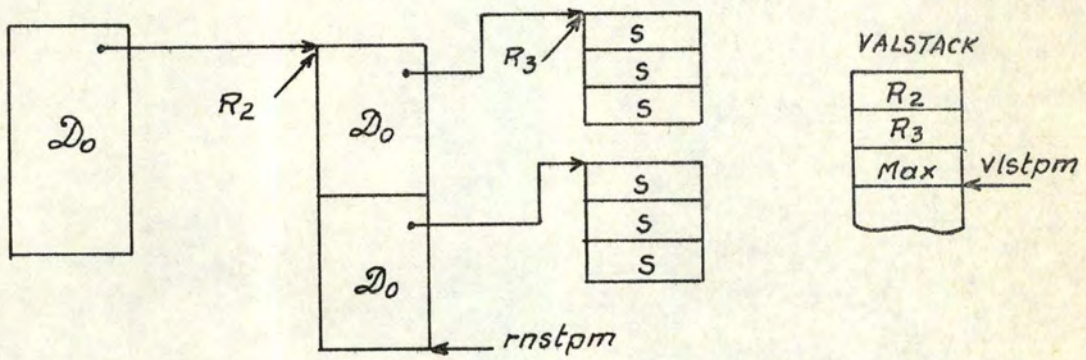


D_o : Descripteur objet
o : Elément objet

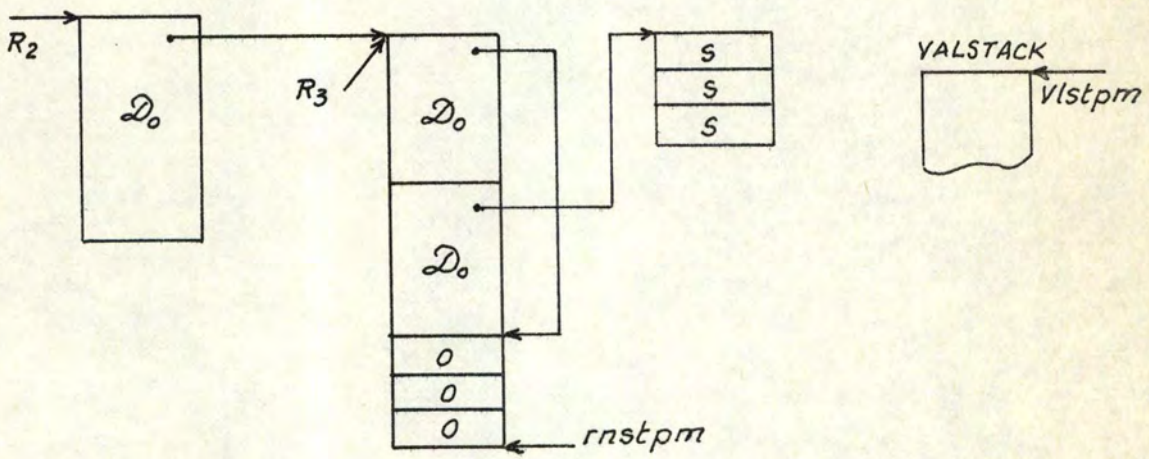
③ Valeur après la copie des parties statiques des éléments et la mise à jour du pointeur



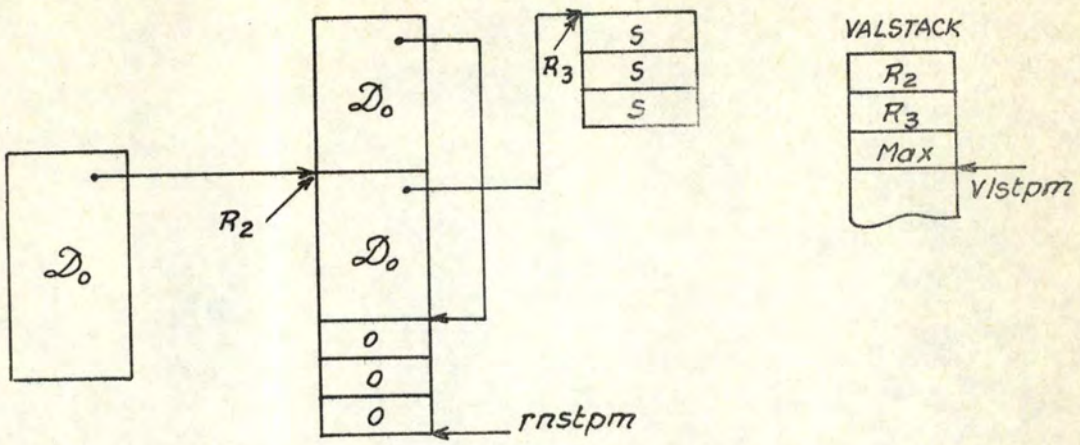
④ Valeur après la préparation de la copie de la partie dynamique du 1° élément



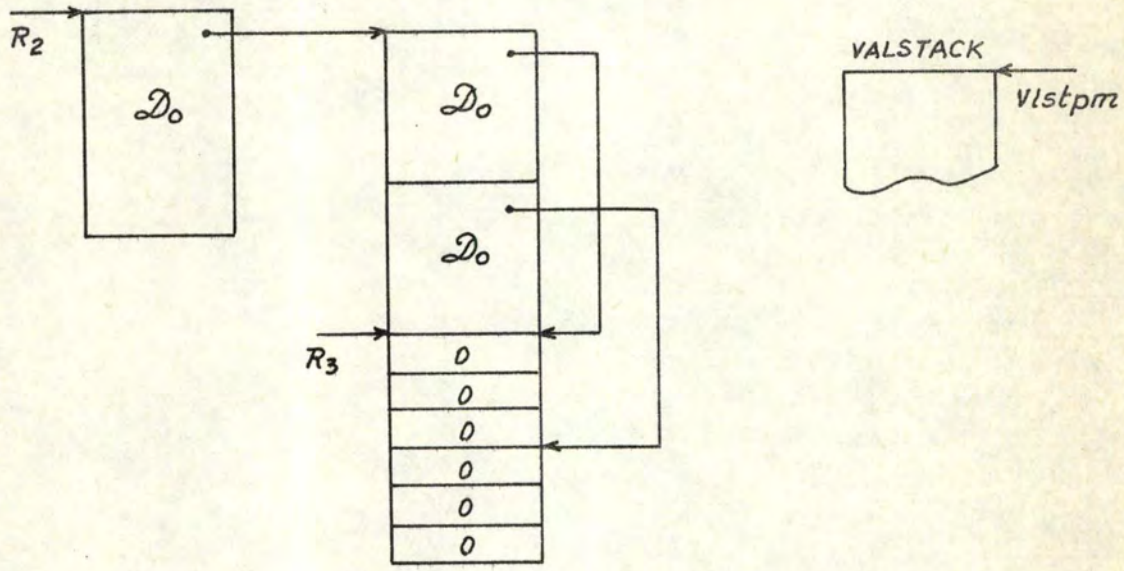
⑤ Valeur après la copie envisagée en ④



⑥ Valeur après la préparation de la copie de la partie dynamique du 2° élément



⑦ Valeur après la copie envisagée en ⑥



TEXTE PL/I

```
RØWØF : PRØC (ADD, MØDE);
     CALL MØDERW (MØDE, MØDER) ;          /* moder est le mode d'un */
                                          /* élément                */

     CALL STSIZE (MØDER, STATSIZE) ;
     CALL RELVNT (MØDER, DYNREL, DYNSCØPE);
     CALL MODNMB (MØDE, NDIM) ; /* nombre de dimensions de la valeur */
                                /* = NDIM                            */
```

GEN

```
Calcul de place disponible

IF FLAG = 1        /* test de la continuité des éléments   */
   THEN GØTØ L1;
   ELSE DØ ; Calcul de d
                                                          o
             D(N) = STATSIZE ;
             DØ I = N-1 TØ 0 BY-1 ;
               D(I) = (U(I+1)-L(I+1)+1)*D(I+1);
             END;
             Sauvetage des éléments d  sur VALSTACK
                                     i
        END;
L1:IF (heappm - rnstpm) < d
                          o
     THEN CALL GARBAGE-CØLLECTØR ;
```

GEN

```
IF FLAG = 1
  THEN GØTØ LCONS ;
```

/* génération de code */
/* pour les éléments */
/* consécutifs */

```
CALL INITAL (NDIM) ;
CALL COPYST (R3,R11,STATSIZE);
```

/* copie des éléments */
/* s'il ne sont pas */
/* consécutifs. */

GEN

```
rnstpm = rnstpm + STATSIZE
```

/* mise à jour du nom- */
/* met de la pile */

```
CALL CØNTIN (NDIM) ;
```

GEN

```
Mise à jour du descripteur objet à
l'aide des éléments retirés de
VALSTACK
GØTØ LD ;
```

GEN

```
L CØNS :
```

```
CALL CØPYD2 (R3,R11) ;
```

/* copie des éléments */
/* s'il sont consécu- */
/* tifs */

```
GEN
```

POINTEUR = RNSTPM : mise à jour du pointeur dans le descripteur objet
RNSTPM = RNSTPM + d_o : mise à jour du sommet de la pile

```
GEN
```

LD :

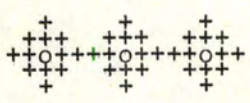
```
;

/* ADD1=0      ADD1 est une adresse relative, elle représente   */
/*             le déplacement de l'élément traité par rapport   */
/*             au premier élément de la valeur.                 */

IF DYNREL  =  '1'B | DYNSCØPE ='1'B

  THEN DØ ; GEN
```

MAX = RNSTPM	la valeur max est la valeur de comparaison pour voir si on a copié tous les éléments.
LDYN: Sauver R2,R3, Max sur VALSTACK Mise à jour de vlstpm	
M [RE] = M [R3] (1)	passer à travers des descripteurs

```
       CALL CPDNT (ADD1,MØDER) ;
       /* ADD1 = ADD1 + STATSIZE      */

       GEN
```

Restituer R2, R3 et MAX à partir de VALSTACK mise à jour de vlstpn
M[R3] = M[R] + STATSIZE.Mise à jour de R3 IF M[R3]<MAX
THEN GØTØ LDYN

```
     END ;
  END RØWØF ;
```



(1) M [R3] signifie le contenu du régistre R3

CHAPITRE 4 : CONCLUSIONS

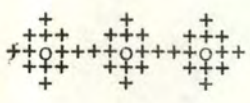
On peut dire, qu'à l'occasion de ce mémoire on a largement atteint le but fixé.

Le but principal était de décrire un algorithme de génération de code qui correspond à la copie d'une valeur pendant l'exécution du programme. Nous ne l'avons pas simplement décrit, mais nous l'avons expliqué au moyen d'une grammaire générative. Cette grammaire nous permet en ayant un arbre grammatical devant soi de voir ce qui se passe pendant la compilation (l'arbre lui-même) et ensuite de voir ce qui se passe à l'exécution (chaîne terminale de sortie). La description par la voie d'une grammaire nous permet de voir ce qui se passe tout en restant clair et précis, mais n'entrant tout de même pas dans des détails techniques liés à la machine.

Le but suivant était de décrire l'algorithme en PL/I et de générer du code assembleur IBM360. Nous l'avons fait, et nous avons même crée un environnement qui servira de base à une implémentation future d'un compilateur ALGOL68 au CTI/ERM. Nous avons vu que le PL/I admet une génération de code aisée et que ce langage pourrait se montrer apte à l'écriture ou du moins à la documentation d'un compilateur. En écrivant cet algorithme dans ce langage de haut niveau, on sait qu'il reste pratiquement indépendant de la machine. Sa liaison se fait par certains appels de routine comme ADRIBM, CØPYST, CØPYD2, SAVREG, RESREG.

En plus que la description d'un algorithme de copie, on a utilisé différentes techniques de compilation comme la génération de code, le parcours de valeurs multiples et la génération de tables de branchements (qui peuvent servir à la génération ultérieure des tables au cas d'instructions if then else fi) en dernier lieu on a fait un générateur d'étiquettes.

Le mémoire n'a pas été alourdi inutilement par la description en détail des procédures PL/I. On les retrouve en annexe.



A N N E X E S

a. Algorithme complet.

Nous trouvons ci-dessous le texte complet des routines décrites dans ce mémoire tel qu'elles sont implémentées à l'ERM. Dans ces routines on tient compte de l'alignement, c'est une contrainte IBM360. Nous n'en avons pas parlé, car cela aurait alourdi inutilement le problème. Les routines se trouvent dans le même ordre qu'elles ont été décrites dans le mémoire. Le texte PL/I se trouve en page 2 et les suivantes.

b. Remarques.

On peut faire des optimisations à notre génération de code.

(1) Un deuxième passage sera nécessaire pour éliminer les instructions inutiles.

```
Exemple :    L     R1, addr
             ST    R1, addr
```

Ces instructions doivent être éliminées.

(2) Nous générons du code pour chaque valeur qui apparaît dans le programme. Cela n'est pas efficace. Il vaut mieux de ne générer qu'un code de copie par mode apparant et de brancher vers ce code à chaque occurence d'une copie de ce mode. Cette optimisation a été mise au point à l'ERM. L'information nécessaire pour savoir si une routine a déjà été générée ou non se trouve dans la table des modes. Les instructions nécessaires à cette optimisation ne se trouvent pas dans le texte.

(3) Au lieu de garder des tables partielles de code dans les routines PL/I on aurait pu concevoir une table qui contient tout le code. A l'appel GEN on aurait pris une partie de la table qui correspond au code voulu.

```
GEN:PROC (GENER,NUMB);

      DCL GENER(26) CHAR(40);
      DCL IN  CHAR(40);
      DCL OUT CHAR(80);
      DCL ROWIN(40)  CHAR(1) DEF IN;
      DCL ROWOUT(80) CHAR(1) DEF OUT;
      DCL J,L BIN FIXED;
      DCL NUMB DEC FIXED(2,0);

TRNFRT:  PROC;  /* TRANSFERT DE CARACTERES */
           DO WHILE (ROWIN(J) ¬= ' ');
             ROWOUT(L)=ROWIN(J);
             J=J+1;
             L=L+1;
           END;
END TRNFRT;

SKPBLK:  PROC;  /* SAUTER LES BLANCS */
           DO WHILE (ROWIN(J)  = ' ');
             J=J+1;
           END;
END SKPBLK;

                    /* PROCEDURE GEN MEME */

      DO I=1 TO NUMB;
        OUT=' ';
        IN=GENER(I);
        J=1;
        IF ROWIN(1) ¬= ' '
          THEN DO;L=1;
                 CALL TRNFRT;
               END;
        L=16;
        J=J+1;
        CALL SKPBLK;
        CALL TRNFRT;
        L=16;
        J=J+1;
        CALL SKPBLK;
        CALL TRNFRT;
        PUT SKIP LIST (OUT);
      END;
     PUT SKIP LIST('*-----------------');
 END GEN;
```

```
RELVNT: PROC(MODE,DYNREL,DYNSCOPE);
   DCL MODE BIN FIXED,
       DYNREL BIT(1),
       DYNSCOPE BIT(1);
   DCL SORT BIN FIXED,
       STAP DEC FIXED(1),
       MODTAB(30) CHAR(24) EXT,
       DESTAB(110) BIN FIXED EXT,
       BUF CHAR(24) DEF BUFR,
       1 BUFR, 2 ADDR BIN FIXED,
               2 NIL  CHAR(20);
   BUF=MODTAB(MODE);
   DYNSCOPE='0'B; DYNREL='0'B;
   IF ADDR=0 THEN RETURN;
   SORT=DESTAB(ADDR);
   IF SORT=-4 THEN DO;CALL UNISTR(2);
                      RETURN;
                   END;
   IF SORT=-5 THEN DO;CALL UNISTR(1);
                      RETURN;
                   END;
   IF SORT=-6 THEN DO;DYNREL='1'B;
                      RETURN;
                   END;
   UNISTR: PROC(STAP);
     DCL (FIRST,LAST) BIN FIXED,
          STAP   DEC FIXED(1);
     FIRST=ADDR+2;
     LAST=ADDR+DESTAB(ADDR+1)*STAP;
     IF STAP=1 THEN LAST=LAST+1;
     DO I=FIRST BY STAP TO LAST;
          BUF=MODTAB(DESTAB(I));
          SORT=DESTAB(ADDR);
          IF SORT=-6 THEN DYNREL='1'B;
          IF SORT<0 & SORT>-4 THEN DYNSCOPE='1'B;
     END;
     RETURN;
   END UNISTR;
   DYNSCOPE='1'B;
END RELVNT;
```

```
STSIZE: PROC (MODE,STATSIZE);
   DCL  MODTAB(30)  CHAR(24) EXT,
        DESTAB(110) BIN FIXED EXT,
        BUF CHAR(24) DEF BUFR,
        1 ALLIN, 2 MULT,
                 2 NREST,
        1 BUFR, 2 ADDR BIN FIXED,
                2 ALL1 BIN FIXED,
                2 ALL2 BIN FIXED,
                2 LABI BIN FIXED,
                2 NIL CHAR(4),
                2 STAT BIN FIXED;
   DCL (MODE,STATSIZE,MODETYPE,MODEF,MODEU,MODER,NUMB,I) BIN FIXED;
   DCL (MODUL,MODIF) BIN FIXED;
   BUF=MODTAB(MODE);
   STATSIZE=STAT;
   RETURN;
MODTYP: ENTRY(MODE,MODETYPE);
   BUF=MODTAB(MODE);
   MODETYPE=-DESTAB(ADDR);
   RETURN;
MODNXT: ENTRY(I,MODE,MODEF);
   BUF=MODTAB(MODE);
   MODEF=DESTAB(ADDR+2*I);
   RETURN;
MODUNXT: ENTRY(I,MODE,MODEU);
   BUF=MODTAB(MODE);
   MODEU=DESTAB(ADDR+I+1);
   RETURN;
MODERW: ENTRY(MODE,MODER);
   BUF=MODTAB(MODE);
   MODER=DESTAB(ADDR+2);
   RETURN;
MODNMB: ENTRY(MODE,NUMB);
   BUF=MODTAB(MODE);
   NUMB=DESTAB(ADDR+1);
   RETURN;
ALLIGN: ENTRY(MODE,ALLIN);
   BUF=MODTAB(MODE);
   MULT=ALL1;
   NREST=ALL2;
   RETURN;
MODLAB: ENTRY(MODE,MODUL);
   BUF=MODTAB(MODE);
   MODUL=LABI;
   RETURN;
MODIFL: ENTRY(MODE,MODIF);
   BUF=MODTAB(MODE);
   LABI=MODIF;
   MODTAB(MODE)=BUF;
END STSIZE;
```

```
ADRIBM: PROC(REG,ADD);
   DCL REG BIN FIXED;
   DCL 1 ADD, 2 N BIN FIXED,
              2 ND BIN FIXED;
   DCL GEN ENTRY,
       TEXT(2) CHAR(40),
       NR PIC '99',
       N1 PIC '9999';
   NR=REG; N1=ND;
   IF N=-1  THEN DO;TEXT(1)=' LA '||NR||',0('||N1||'('||NR||')';
                    CALL GEN(TEXT,01);
                    RETURN;
                 END;
   IF N=0   THEN DO;TEXT(1)=' LA '||NR||',0('||N1||'(13)';
                    CALL GEN(TEXT,01);
                    RETURN;
                 END;
   IF N>=1  THEN DO;N1=4*(18+N);
                    TEXT(1)=' L '||NR||',0('||N1||'(13)';
                    N1=ND;
                    TEXT(2)=' LA '||NR||',0('||N1||'('||NR||')';
                    CALL GEN(TEXT,02);
                    RETURN;
                 END;
   IF N=-3 THEN DO;TEXT(1)=' LR   2,3';
                   CALL GEN(TEXT,01);
                   RETURN;
                END;
   IF N=-2  THEN DO;TEXT(1)=' L '||NR||',CONSTAB(12)';
                    TEXT(2)=' LA '||NR||',0('||N1||'('||NR||')';
                    CALL GEN(TEXT,02);
                 END;
END ADRIBM;
```

```
COPYST: PROC(RS,RO,SIZE);
   DCL  GEN ENTRY,
        (RS,RO,SIZE) BIN FIXED,
        GENER(26) CHAR(40) EXT,
       (NRS,NRO) PIC '99',
        NSZ  PIC '9999';
   NRS=RS; NRO=RO; NSZ=SIZE;
   IF SIZE <= 256
     THEN DO;GENER(1)=' MVC 0('||NSZ||',0('||NRO||'),0('||NRS||')';
             CALL GEN(GENER,01);
          END;
     ELSE BEGIN;
            DCL (ST,IMOD) BIN FIXED;
            ST=SIZE/256;
            GENER(1)=' LR   6,'||NRS;
            GENER(2)=' LR   7,'||NRO;
            NSZ=ST;
            GENER(3)=' LA   8,'||NSZ||'(0)';
            GENER(4)=' MVC  0(256,7),0(6)';
            GENER(5)=' LA   6,256(6)';
            GENER(6)=' LA   7,256(7)';
            GENER(7)=' BCT  8,*-14';
            CALL GEN(GENER,07);
            IMOD=MOD(SIZE,256);
            IF IMOD ¬= 0
              THEN DO;NSZ=IMOD;
                      GENER(1)=' MVC 0('||NSZ||',7),0(6)';
                      CALL GEN(GENER,01);
                   END;
          END;
END COPYST;
```

```
SAVREG: PROC(N,NREG);
   DCL NG PIC '99',
       GENER(26) CHAR(40) EXT,
       GEN ENTRY;
   NG=NREG;
   IF N¬=-1 THEN RETURN;
   GENER(1)=' L     6,VLSTPM';
   GENER(2)=' ST '||NG||',0(6)';
   GENER(3)=' LA    6,4(6)';
   GENER(4)=' ST    6,VLSTPM';
   CALL GEN(GENER,04);
END SAVREG;
```

```
RESREG: PROC(N,NREG);
   DCL NG PIC '99',
       GENER(26) CHAR(40) EXT,
       GEN ENTRY;
   NG=NREG;
   IF N¬=-1 THEN RETURN;
   GENER(1)=' L     6,VLSTPM';
   GENER(2)=' S     6,=F''4''';
   GENER(3)=' L '||NG||',0(6)';
   GENER(4)=' ST    6,VLSTPM';
   CALL GEN(GENER,04);
END RESREG;
```

```
COPYD2: PROC(RS,RD);
   DCL (RS,RD) BIN FIXED,
       (NRS,NRD) PIC '99',
        GEN ENTRY,
        GENER(15) CHAR(40) INIT
         ((2)(1)' ',
          ' LH     8,0(2)',
          ' CH     8,=H''256''',
          ' BH     *+22',
          ' BCT    8,0',
          ' STC    8,*+5',
          ' MVC    0(0,7),0(6)',
          ' B      *+30',
          ' MVC    0(256,7),0(6)',
          ' LA     7,256(7)',
          ' LA     6,256(6)',
          ' SH     8,=H''256''',
          ' CH     8,=H''0''',
          ' BNE    *-48');
   NRS=RS;  NRD=RD;
   GENER(1)=' LR    6,'||NRS;
   GENER(2)=' LR    7,'||NRD;
   CALL GEN(GENER,15);
END COPYD2;
```

```
INITAL: PROC(N);
   DCL N BIN FIXED,
       GEN ENTRY,
       LAB BIN FIXED EXTERNAL,
       N1  PIC '9999',
       GENER(24) CHAR(40) INIT
        (' LA     6,VLSTPM',
         ' LA     7,8(2)',
         ' ',
         ' MVC    0(2,6),0(7)',
         ' LA     6,4(6)',
         ' LA     7,6(7)',
         ' BCT    8,*-14',
         (3)(1)' ',
         ' MVC    0(2,6),4(7)',
         ' LA     5,1(0)',
         ' AH     5,2(7)',
         ' SH     5,0(7)',
         ' MH     4,4(7)',
         ' LNR    5,5',
         ' AH     5,0(6)',
         ' SH     6,=H''4''',
         ' SH     7,=H''6''',
         ' AH     5,4(7)',
         ' STH    5,0(6)',
         ' BCT    8,*-40',
         ' MVI    4(2),=X''FF''',
         ' ');
    IF N=1
      THEN BEGIN;
             DCL TEXT(5) CHAR(40) INIT
                   (' MVI  4(2),=X''FF''',
                    ' LH     8,10(2)',
                    ' SH     8,8(2)',
                    ' LA     8,1(8)');
             N1=LAB;
             TEXT(5)='L'||N1||' EQU *';
             CALL GEN(TEXT,05);
             RETURN;
           END;
    N1=N;
    GENER( 3)=' LA     8,'||N1||'(0)';
    N1=4*(N-1);
    GENER( 8)=' LA     6,'||N1||'(6)';
    N1=6*N+2;
    GENER( 9)=' LA     7,'||N1||'(2)';
    N1=N-1;
    GENER(10)=' LA     8,'||N1||'(0)';
    N1=LAB;
    GENER(24)='L'||N1||' EQU *';
    CALL GEN(GENER,24);
END INITAL;
```

```
CONTIN: PROC(N);
   DCL N BIN FIXED;
   DCL GEN  ENTRY,
       GENER(26) CHAR(40) EXTERNAL,
       LAB  BIN FIXED  EXTERNAL;
   DCL N1 PIC '9999';
   IF N=1
     THEN DO;GENER(1)=' AH   3,12(2)';
              N1=LAB;  LAB=LAB+1;
              GENER(2)=' BCT  8,L'||N1;
              CALL GEN(GENER,2);
              RETURN;
          END;
   N1=4*(N-1);
   GENER(01)=' LA     6,VLSTPM+'||N1;
   N1=2*N;
   GENER(02)=' LA     7,'||N1||'(6)';
   N1=N;
   GENER(03)=' LA     8,'||N1||'(0)';
   N1=6*N+2;
   GENER(04)=' LA     9,'||N1||'(2)';
   GENER(05)=' CLC    0(2,6),2(9)';
   GENER(06)=' BNE    *+14';
   GENER(07)=' MVC    0(2,6),0(9)';
   GENER(08)=' B      *+24';
   GENER(09)=' LH     4,0(6)';
   GENER(10)=' LA     4,1(4)';
   GENER(11)=' STH    4,0(6)';
   GENER(12)=' AH     3,0(7)';
   N1=LAB; LAB=LAB+1;
   GENER(13)=' B      L'||N1;
   GENER(14)=' SH     6,=H''4''';
   GENER(15)=' SH     7,=H''2''';
   GENER(16)=' SH     9,=H''6''';
   GENER(17)=' BCT    8,*-52';
   CALL GEN(GENER,17);
END CONTIN;
```

```
COPY:    PROC(AD1,AD2,MODE);
   DCL (STSIZE,RELVNT,ADRIBM,COPYST,CPDNPT,GEN) ENTRY,
        GENER(26) CHAR(40) EXT,
        1 AD1, 2 NAD1,
               2 NDEP1,
        1 AD2, 2 NAD2,
               2 NDEP2,
       (MODE,STATSIZE) BIN FIXED,
        R2 BIN FIXED INIT(2),
        R3 BIN FIXED INIT(3),
        DYNREL BIT(1),
        DYNSCOPE BIT(1),
       (NPROC,DPROC,NBLOC,DBLOC) BIN FIXED EXT;
   PUT SKIP LIST('*----- STAT -------');
   CALL STSIZE(MODE,STATSIZE);
   CALL ADRIBM(R2,AD1);
   CALL ADRIBM(R3,AD2);
   CALL COPYST(R2,R3,STATSIZE);
   IF STATSIZE <= 4 THEN RETURN;
   CALL RELVNT(MODE,DYNREL,DYNSCOPE);
   IF DYNREL = '0'B & DYNSCOPE = '0'B
      THEN DO; PUT SKIP LIST('*----- ENDFILE ----');
              RETURN;
           END;
   IF NAD2=0
     THEN NPROC=DPROC;
     ELSE NPROC=NAD2;
   NBLOC=DBLOC;
   NAD2=-3;
   CALL CPDNPT(AD2,MODE);
   PUT SKIP LIST('*----- ENDFILE ----');
END COPY;
```

```
CPDNPT: PROC(ADD,MODE);
   DCL (STSIZE,MODTYP,STRUCT,UNION,ROWOF,GEN) ENTRY,
       (SAVREG,RESREG,ADRIBM)ENTRY,
        R2 BIN FIXED INIT(2),
        1 ADD, 2 N,
               2 ND,
       (MODEX,MODE) BIN FIXED,
       (NPROC,DBLOC) BIN FIXED EXT,
        DYNREL    BIT(1),
        DYNSCOPE BIT(1),
        GENER(26) CHAR(40) EXT,
        NG PIC '9999';
   CALL MODTYP(MODE,MODEX);
     IF MODEX >= 1 & MODEX <= 3
                THEN DO;NG=NPROC;
                        GENER(1)=' LA    5,'||NG||'(0)';
                        GENER(2)=' SLL   5,16';
                        NG=DBLOC;
                        GENER(3)=' LA    6,'||NG||'(0)';
                        GENER(4)=' AR    5,6';
                        IF N¬=-1 THEN NG=0;
                                 ELSE NG=ND;
                        GENER(5)=' CL    5,'||NG||'(2)';
                        GENER(6)=' BL    SCOPEERR';
                        PUT SKIP LIST('*----- SCOPE -----');
                        CALL GEN(GENER,06);
                        RETURN;
                      END;
               CALL SAVREG(N,R2);
               CALL ADRIBM(R2,ADD);
               IF MODEX=4
                 THEN CALL STRUCT(ADD,MODE);
               IF MODEX=5
                 THEN CALL UNION(ADD,MODE);
               IF MODEX=6
                 THEN CALL ROWOF(ADD,MODE);
               CALL RESREG(N,R2);
END CPDNPT;
```

```
STRUCT: PROC(ADD,MODE);
   DCL (ALLIGN,MODNMB,MODNXT,RELVNT,CPDNPT,STSIZE) ENTRY,
        1 ADD1,2 N1,
                2 ND1,
        1 ALLIN, 2 MULT,
                 2 NREST,
       (STATSIZE,NUMFIELDS,MODE,MODEF) BIN FIXED,
        DYNREL    BIT(1),
        DYNSCOPE BIT(1);
   PUT SKIP LIST('*----- STRBEG -----');
   CALL MODNMB(MODE,NUMFIELDS);
   CALL ALLIGN(MODE,ALLIN);
   N1=-1;  ND1=0;  ND2=NREST;  NDELTA=NREST;
   DO I=1 TO NUMFIELDS;
      CALL MODNXT(I,MODE,MODEF);
      CALL RELVNT(MODEF,DYNREL,DYNSCOPE);
      CALL STSIZE(MODEF,STATSIZE);
      CALL ALLIGN(MODEF,ALLIN);
      NMOD=MOD(ND2,MULT);
      IF NMOD < NREST THEN ND1=ND1+NREST-NMOD;
      IF NMOD > NREST THEN ND1=ND1+MULT+NREST-NMOD;
      IF DYNREL = '1'B | DYNSCOPE = '1'B
         THEN CALL CPDNPT(ADD1,MODEF);
         ELSE PUT SKIP LIST('*----- NIL ------');
      ND1=ND1+STATSIZE;
      ND2=ND1+NDELTA;
   END;
   PUT SKIP LIST('*----- STREND -----');
END STRUCT;
```

```
UNION:  PROC(ADD,MODE);
   DCL (ALLIGN,MODNMB,MDUNXT,RELVNT,CPDNPT,GEN) ENTRY,
        GENER(26) CHAR(40) EXT,
        1 ADD1,2 N1,
                2 ND1,
        1 ALLIN, 2 MULT,
                 2 NREST,
       (MODE,MODEU,NUMMODE) BIN FIXED,
        DYNREL    BIT(1),
        DYNSCOPE BIT(1),
        LUN PIC '9999',
        NG  PIC '99',
        LAB BIN FIXED EXT;
   PUT SKIP LIST('*----- UNBEG ------');
   LUN=LAB; LAB=LAB+1;
   CALL MODNMB(MODE,NUMMODE);
   GENER(1)=' LA   7,*+8';
   GENER(2)=' AH   7,0(2)';
   GENER(3)=' BR   7';
   CALL GEN(GENER,03);
   CALL ALLIGN(MODE,ALLIN);
   INREST=NREST+2;
   DO I=1 TO NUMMODE;
     NG=I;
     CALL MDUNXT(I,MODE,MODEU);
     CALL RELVNT(MODEU,DYNREL,DYNSCOPE);
     IF DYNREL ='1'B | DYNSCOPE = '1'B
       THEN GENER(1)=' B LU'||NG||LUN;
       ELSE GENER(1)=' B LUOO'||LUN;
     CALL GEN(GENER,01);
   END;
   DO I=1 TO NUMMODE;
     NG=I; REL=0;
     CALL MDUNXT(I,MODE,MODEU);
     CALL RELVNT(MODEU,DYNREL,DYNSCOPE);
     IF DYNREL = '1'B | DYNSCOPE = '1'B
       THEN DO;GENER(1)='LU'||NG||LUN||' EQU *';
               CALL GEN(GENER,01);
               N1=-1; ND1=2;
               CALL ALLIGN(MODEU,ALLIN);
               NMOD=MOD(INREST,MULT);
               IF NMOD < NREST THEN ND1=ND1+NREST-NMOD;
               IF NMOD > NREST THEN ND1=ND1+MULT+NREST-NMOD;
               CALL CPDNPT(ADD1,MODEU);
               GENER(1)=' B LUOO'||LUN;
               CALL GEN(GENER,01);
            END;
       ELSE PUT SKIP LIST('*----- NIL ------');
   END;
   GENER(1)='LUOO'||LUN||' EQU *';
   CALL GEN(GENER,01);
   PUT SKIP LIST('*----- UNEND ------');
END UNION;
```

```
ROWOF:  PROC(ADD,MODE);
   DCL (MODERW,RELVNT,STSIZE,INITAL,COPYST,CONTIN,COPYD2,
        MODNMB,CPDNPT,GEN,ALLIGN) ENTRY,
        GENER(26) CHAR(40) EXT,
       (LCONS,LD,LRDYN,NG) PIC '9999',
        LAB  BIN FIXED EXT,
        1 ADD1, 2 N1,
                2 ND1,
        1 ALLIN, 2 MULT,
                 2 NREST,
       (MODE,MODER,STATSIZE,NDIM)  BIN FIXED,
        DYNREL    BIT(1),
        DYNSCOPE BIT(1),
        R3 BIN FIXED  INIT(3),
        R11 BIN FIXED INIT(11);
   PUT SKIP LIST('*----- ROWBEG -----');
   PUT SKIP LIST('*----- STATROW ----');
   CALL MODERW(MODE,MODER);
   CALL STSIZE(MODER,STATSIZE);
   CALL RELVNT(MODER,DYNREL,DYNSCOPE);
   CALL MODNMB(MODE,NDIM);
   CALL ALLIGN(MODER,ALLIN);
   NG=MULT;
   GENER(1)=' LA    4,'||NG;
   NG=NREST;
   GENER(2)=' LA    5,'||NG;
   GENER(3)=' BAL   14,ALLIGN';
   CALL GEN(GENER,03);
   NMOD=MOD(STATSIZE+NREST,MULT);
   IF NMOD < NREST THEN STATSIZE=STATSIZE+NREST-NMOD;
   IF NMOD > NREST THEN STATSIZE=STATSIZE+MULT+NREST-NMOD;
   DCL  TEXT(26) CHAR(40) INIT
       (' TM     4(2),X''FF''',
        ' BC     8,*+66',
        ' L      5,VLSTPM',
         (2)(1)' ',
        ' STH    7,0(5)',
        ' LA     5,4(5)',
        ' ',
        ' LH     6,2(9)',
        ' SH     6,0(9)',
        ' LA     6,1(6)',
        ' MR     6,6',
        ' STH    7,0(5)',
        ' LA     5,4(5)',
        ' S      9,=F''6''',
        ' BCT    8,*-26',
        ' ST     5,VLSTPM',
        ' B      *+8',
        ' LH     7,6(2)',
        ' L      6,HEAPPM',
        ' L      3,0(2)',
        ' AR     7,11',
        ' CR     7,6',
        ' BNH    *+12',
        ' L      15,=A(GARBAGE)',
        ' BALR   14,15');
    NG=6*NDIM+2;
    TEXT( 4)='  LA 9,'||NG||'(2)';
    NG=STATSIZE;
    TEXT( 5)=' LA  7,'||NG||'(3)';
```

Ann 14.-

```
NG=NDIM;
TEXT( 8)=' LA   8,'||NG||'(0)';
CALL GEN(TEXT,26);
LCONS=LAB; LAB=LAB+1;
GENER(1)=' ST   11,0(2)';
GENER(2)=' TM   4(2),X''FF''';
GENER(3)=' BC   8,L'||LCONS;
CALL GEN(GENER,03);
CALL INITAL(NDIM);
CALL COPYST(R3,R11,STATSIZE);
NG=STATSIZE;
GENER(1)=' LA    11,'||NG||'(11)';
CALL GEN(GENER,01);
CALL CONTIN(NDIM);
LD=LAB;  LAB=LAB+1;
GENER(1)=' L     6,VLSTPM';
GENER(2)=' LA    7,6(2)';
NG=NDIM+1;
GENER(3)=' LA   8,'||NG||'(0)';
GENER(4)=' S    6,=F''4''';
GENER(5)=' MVC  0(2,7),0(6)';
GENER(6)=' LA   7,6(7)';
GENER(7)=' BCT  8,*-14';
GENER(8)=' L     3,0(2)';
GENER(9)=' B   L'||LD;
GENER(10)='L'||LCONS||' EQU *';
CALL GEN(GENER,10);
CALL COPY02(R3,R11);
GENER(1)=' LR     3,11';
GENER(2)=' AH     11,6(2)';
GENER(3)='L'||LD||' EQU *';
CALL GEN(GENER,03);
IF DYNREL = '1'B | DYNSCOPE = '1'B
   THEN DO;N1=-1; ND1=0;
          PUT SKIP LIST('*----- LOOP ------');
           LRDYN=LAB; LAB=LAB+1;
           GENER(1)=' L  6,VLSTPM';
           GENER(2)=' LR 4,11';
           GENER(3)='L'||LRDYN||' STM 2,4,0(6)';
           GENER(4)=' LA 6,12(6)';
           GENER(5)=' ST 6,VLSTPM';
           GENER(6)=' LR 2,3';
           CALL GEN(GENER,06);
           CALL CPDNPT(ADD1,MODER);
           GENER(1)=' L    6,VLSTPM';
           GENER(2)=' S    6,=F''12''';
           GENER(3)=' ST   6,VLSTPM';
           GENER(4)=' LM   2,4,0(6)';
           NG=STATSIZE;
           GENER(5)=' LA   3,'||NG||'(3)';
           GENER(6)=' CR   3,4';
           GENER(7)=' BL   L'||LRDYN;
           CALL GEN(GENER,07);
        END;
  PUT SKIP LIST('*----- ROWEND -----');
END ROWOF;
```

c. Exemple.

Nous trouvons successivement le code qui peut exécuter la copie d'une valeur de mode int, long int, ref real, union (int, real) et de mode [] real.

Nous donnons aussi la MODTAB et la DESTAB correspondante.

Chaque code généré commence par l'information suivante.

NAD1	=	n source	(n,p)
NDEP1	=	p source	
NAD2	=	n objet	(n,p)
NDEP2	=	p objet	
MODE	=	entrée dans MODTAB	

Remarque : si n=0 nous voulons dire que nous travaillons dans la procédure qui se trouve au sommet de la pile. (C'est une optimisation).

MODTAB

ADDR	ALL1	ALL2	LAB1	NIL	STAT	
0	2	0	0		2	← int
0	4	0	0		4	← long int
0	4	0	0		4	
1	4	0	0		8	
3	4	0	0		8	← ef real
5	4	2	0		6	
11	4	2	0		6	← union (int, real)
15	4	0	0		14	← [] real
18	4	0	0		16	
21	4	2	0		24	
29	4	2	0		22	
35	4	0	0		20	
38	4	0	0		14	
41	4	0	0		14	
44	4	2	0		24	
52	4	0	0		14	
55	4	0	0		14	
58	4	0	0		20	
61	4	2	0		28	
67	4	0	0		26	
70	4	0	0		14	
73	4	0	0		28	
76	4	0	0		14	
82	4	2	0		16	
90	4	0	0		14	
93	4	2	0		16	

DESTAB

-1	1	-1	3	-4	2
1	0	3	0	-5	2
1	3	-6	1	3	-2
0	0	-4	3	1	0
8	0	4	0	-5	4
1	12	3	13	-6	2
3	-6	1	1	-6	1
15	-4	3	1	0	5
0	16	0	-6	1	6
-6	1	18	-6	3	19
-4	2	1	0	20	0
-6	3	21	-6	1	22
-4	2	8	0	13	0
-6	1	24	-4	3	1
0	25	0	21	0	-6
1	26	-5	2	1	13

Code Assembleur IBM360

int

```
NAD1=     7
NDEP1=  120
NAD2=     4
NDEP2=  176
MODE=     1
```

```
*----- STAT -------
          L       02,0100(13)
          LA      02,0120(02)
*------------------
          L       03,0088(13)
          LA      03,0176(03)
*------------------
          MVC     0(0002,03),0(02)
*------------------
```

---

long int

```
NAD1=     8
NDEP1=  130
NAD2=     8
NDEP2=1888
MODE=     2
```

```
*----- STAT -------
          L       02,0104(13)
          LA      02,0130(02)
*------------------
          L       03,0104(13)
          LA      03,1888(03)
*------------------
          MVC     0(0004,03),0(02)
*------------------
```

---

ref real

```
NAD1=     0
NDEP1=  170
NAD2=     2
NDEP2=  180
MODE=     5
```

```
*----- STAT -------
          LA      02,0170(13)
*------------------
          L       03,0080(13)
          LA      03,0180(03)
*------------------
          MVC     0(0008,03),0(02)
*------------------
*----- SCOPE -----
          LA      5,0002(0)
          SLL     5,16
          LA      6,0003(0)
          AR      5,6
          CL      5,0000(2)
          BL      SCOPEERR
*------------------
*----- ENDFILE ----
```

---

union (int, real)

```
NAD1=     0
NDEP1=  150
NAD2=     2
NDEP2=  502
MODE=     7
```

```
*----- STAT -------
          LA      02,0150(13)
*------------------
          L       03,0080(13)
          LA      03,0502(03)
*------------------
          MVC     0(0006,03),0(02)
*------------------
*----- ENDFILE ----
```

```
NAD1=     0
NDEP1=  160
NAD2=     0
NDEP2=  270
MODE=     8
```

[] Rel

```
*----- STAT -------
          LA      02,0160(13)
*------------------
          LA      03,0270(13)
*------------------
          MVC     0(0014,03),0(02)
*------------------
          LR      2,3
*------------------
*----- ROWBEG -----
*----- STATROW ----
          LA      4,0004
          LA      5,0000
          BAL     14,ALLIGN
*------------------
          TM      4(2),X'FF'
          BC      8,*+66
          L       5,VLSTPM
          LA      9,0008(2)
          LA      7,0004(0)
          STH     7,0(5)
          LA      5,4(5)
          LA      8,0001(0)
          LH      6,2(9)
          SH      6,0(9)
          LA      6,1(6)
          MR      6,6
          STH     7,0(5)
          LA      5,4(5)
          S       9,=F'6'
          BCT     8,*-26
          ST      5,VLSTPM
          B       *+8
          LH      7,6(2)
          L       6,HEAPPM
          L       3,0(2)
          AR      7,11
          CR      7,6
          BNH     *+12
          L       15,=A(GARBAGE)
          BALR    14,15
*------------------
          ST      11,0(2)
          TM      4(2),X'FF'
          BC      8,L0001
*------------------
          MVI     4(2),=X'FF'
          LH      8,10(2)
          SH      8,8(2)
          LA      8,1(8)
L0002     EQU     *
*------------------
          MVC     0(0004,11),0(03)
*------------------
          LA      11,0004(11)
*------------------
          AH      3,12(2)
          BCT     8,L0002
*------------------
          L       6,VLSTPM
          LA      7,6(2)
          LA      8,0002(0)
          S       6,=F'4'
          MVC     0(2,7),0(6)
          LA      7,6(7)
          BCT     8,*-14
          L       3,0(2)
          B       L0003
L0001     EQU     *
*------------------
          LR      6,03
          LR      7,11
          LH      8,6(2)
          CH      8,=H'256'
          BH      *+22
          BCT     8,0
          STC     8,*+5
          MVC     0(0,7),0(6)
          B       *+30
          MVC     0(256,7),0(6)
          LA      7,256(7)
          LA      6,256(6)
          SH      8,=H'256'
          CH      8,=H'0'
          BNE     *-48
*------------------
          LR      3,11
          AH      11,6(2)
L0003     EQU     *
*------------------
*----- ROWEND -----
*----- ENDFILE ----
```

BIBLIOGRAPHIE

1. *GRIES D.*
   Compiler construction for digital Computers

2. *BRANQUART P., CARDINAEL J.P., LEWI J.*
   An Optimized Translation Process and its application to ALGOL68 (R204 MBLE)

3. *BRANQUART P. and LEWI J.*
   On the implementation of Local Names in ALGOL68 (R121 MBLE)

4. *BRANQUART P., CARDINAEL J.P., LEWI J.*
   An Opitmized Translation Process, applications to ALGOL68
   (R224 MBLE)

5. *BRANQUART P., CARDINAEL J.P., LEWI J., DELESCAILLE J.P., and VAN BEGIN M.*
   Data Structure handling in ALGOL68 compilation
   (R254 MBLE)

6. *BRANQUART P., LEWI J. and CARDIANEL J.P.*
   Local Generators and the ALGOL68 working Stack
   (TN 62 MBLE)

7. *LINDSEY, VAN DER MEULEN*
   An Informal introduction to ALGOL68

8. *MAILLOUX B.J, PECK J.E.L., KOSTER C.H.A.*
   Report on the algorithmic language ALGOL68

9. *BOURNES.R., BIRREL A.D., WALKER I.*
   ALGOL68 C Reference Manual (University of Cambridge)

10. *GENNART P., LOUIS G.*
    ALGOL72 Reference Manuel (ERM)

11. *BRANQUART P., LEWI J., SINTZOFF and WODON P.L.*
    The composition of Semantics in ALGOL68
    CACM Nov 1971

12. *BRANQUART P. and LEWI J.*
    A Scheme of Storage allocation and Garbage collection for ALGOL68
    (ALGOL Inplementation edited by J.E. L. PECK Noth-Holland)

13. *GRIFFITHS M.*
Run Time Storage Management
Chapitre 3 B (compiler conference Munich 74)

14. *URSULA HILL*
Special Run Time organization techniques for ALGOL68 Chanter 3 C (compiler Conference Munich 74)

15. *BRANQUART P., LOUIS G., WODON P.*
General considerations on storage allocation
Part II Locations and their accesses
Part V Towards the design of a Software programming language (TN 98 MBLE)

16. Assembler 360 Principles of operations

17. IBM System/360 DØS AND TØS PL/I Subset Reference Manual

18. *BUFFER J.*
Traitement des Modes à l'implémentation. Université des Sciences et Techniques de Lille.

19. *KNUTH.*
The arth of Computer Programming
PART II

20. Advanced course on compiler construction
Tech University of Munich March 4-15 (1974)

21. Mode implementatie en ekwivalentie van modes (R31 KMS) (te verschijnen) door *EDDY VAN DE VEL.*

22. Een LR parser generator (R25 KMS)
door *R. HAENTJENS en L. LATHUY*